

A FILING SYSTEM AND SPOOLER FOR THE TDC-316 TIME-SHARING SYSTEM

**A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY**

**by
AARTI KUMAR**

**to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY, 1982**

**Dedicated
To My Brother
R A K E S H**

CENTRAL LIBRARY

Kinder.

Acc. No. **A 82467**

CERTIFICATE

CERTIFIED that the work entitled "A FILING SYSTEM
AND SPOOLER FOR THE TDC-316 TIME-SHARING SYSTEM" by
Aarti Kumar has been carried out under my supervision and
has not been submitted elsewhere for a degree.



A.S. Sethi

Assistant Professor

Computer Science Programme

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Kanpur
July 1982

ABSTRACT

A filing system and a line printer spooler have been designed for TDC-316. Both of them run in the environment of a time sharing system which allows 16 on-line users. Editing of files is envisaged to be the most important activity in the user environment of this system. The filing system is interfaced with the operating system in the shape of twelve primitives. These collectively perform all the filing functions. The filing model has a hierarchical structure which is generalized to allow for its portability to any other operating system designed for TDC-316. The spooler runs as a user process. Other users communicate with it to get their files printed.

ACKNOWLEDGEMENT

I thank Dr. A.S. Sethi, my thesis supervisor for his guidance and encouragement during the course of this work. The regular discussions with him have been very fruitful in completing this project.

I thank Sri S.L. Agarwal and Sri A.K. Basu for their able maintenance of TDC-316 and for the everready help rendered to us.

I thank my project partner Mr. P.K. Pandya for his team spirit and his bright ideas.

I thank George Paul for his suggestions during the designing and invaluable help during the debugging process.

My brother Rakesh Kumar and Mr. Sunil Jha have been a constant source of encouragement. I thank them specially for their much needed assistance in the last few days.

I thank Mr. Mohammed Anwar for his neat and patient typing.

I thank all my friends in the girls hostel and my classmates for their companionship.

Finally I thank all my teachers here for their guidance and help during the course of my M.Tech.

KANPUR
JULY 1982

- AARTI KUMAR

CONTENTS

CHAPTER

1	INTRODUCTION	1
1.1	DESIGN METHODOLOGY AND IMPLEMENTATION	3
1.2	OUTLINE OF THE THESIS	4
2	HARDWARE ENVIRONMENT	5
2.1	THE TDC-316 COMPUTER	5
2.2	THE DISK DRIVE CONTROLLER	7
2.3	THE LINE PRINTER CONTROLLER	9
3	SOFTWARE ENVIRONMENT	10
4	FILING SYSTEM	15
4.1	A HIERARCHICAL OUTLINE	15
4.2	DETAILED DESCRIPTION OF THE SYSTEM	18
5	DISK MANAGEMENT	37
5.1	INTRODUCTION	37
5.2	QUEUE OF REQUESTS	38
5.3	OPERATIONS	39
5.4	INITIATION OF I/O AND INTERRUPT SERVICING	42
6	THE LINE PRINTER SPOOLER	44
6.1	THE SPOOLER PROCESS	44
6.2	LINE PRINTER DRIVER	46

CHAPTER

7	CONCLUSION	48
7.1	ACCOUNTING	48
7.2	ERROR HANDLING	49
7.3	REMARKS	50
	REFERENCES	52
APPENDIX I	LIST OF PRIMITIVES	I
APPENDIX II	ACTIVE TABLE	II
APPENDIX III	LIST OF ERROR MESSAGES	III
APPENDIX IV	PROGRAM LISTINGS	IV

CHAPTER 1

INTRODUCTION

This thesis along with a companion thesis [PAND 82], is concerned with the design and implementation of a small time sharing operating system for the TDC-316. This is a mini-computer of 16 bit wordlength built by the E.C.I.L. Ltd. on the lines of the DEC PDP/11.

Each operating system is designed to serve some specific purpose. Our design was motivated by the needs of a group of users who are new to computer systems and inexperienced programmers and terminal users.

Such a group is best represented by a number of students in an educational institution doing a first course in programming. Their computational efforts are often localized to the editing and running of small programs with a view to learning both the language and terminal usage.

Since our purpose has been to provide such users with adequate facilities, the system as it stands now offers time sharing facilities to a small number of users. (A maximum of 16 on-line users are possible). It also provides users with adequate filing facilities and an extremely fast line oriented test editor. There is also a card reader and line-printer

spooler, allowing the creating of files through cards and the printing of files.

A significant factor in the design is the absence of any language support and hence the facility to execute programs on the machine. This was omitted mainly due to lack of memory in the TDC-316. An alternate provision was made by linking the TDC-316 to the DEC-10 via asynchronous communication line.

Now users can create files on the TDC-316 and submit their programs to the DEC-10 (through the TDC-316) for execution. Since the DEC-10 supports several languages, the programs can be easily executed and the output returned to the TDC-316 through the DEC-TDC link. The output file can be printed by the user on the TDC-316.

Thus the design of the operating system was influenced by the choice of the machine and its inadequacies, the users expected and the linking of the TDC-316 to the DEC-10 which allowed us to omit the execution of programs on the TDC-316 itself.

The design of the filing system was influenced by another factor. It was felt that the filing system should, if it ever be so desired, be transferable to any other operating system written for the TDC-316. Hence its structure has been generalized and its interface with the monitor has been kept relatively simple. The system has a tree structured directory scheme and a protection mechanism which distinguishes between

3 classes of users (a) owner (b) of the same project and (c) others. The interface with the monitor is in the shape of twelve primitives which collectively perform all the filing functions of creating, destroying and modifying files.

1.1 DESIGN METHODOLOGY AND IMPLEMENTATION

The operating system has been designed and implemented in a relatively short time space. Many factors contribute to this. One was the choice of the language BLISS-11 for implementation.

This language was chosen since it was the only system software language available on DEC-10 for use with the TDC-316. The BLISS compiler translated the BLISS code into PDP-11 code. Also available was a cross assembler MACN16 to convert the PDP-11 code to the TDC-316 object code on the DEC-10. This generation of TDC-316 code on the DEC-10 cut down on implementation time. A loader to directly load object programs from the DEC-10 to the TDC-316 via asynchronous lines was also available and hence programs developed on the DEC-10 were directly loaded onto the TDC-316, from where they could be executed directly. There was a linker available to link all programs together and run the combined object code version on the TDC-316.

The design of the operating system was kept relatively simple and carried out in a top-down manner, initially leading from general specifications on what the system is expected to

do to details on how to perform the functions. At the lowest level in the design is the kernel which is the heart of the system and synchronizes and manages all processes and resources.

1.2 OUTLINE OF THE THESIS

The work on the design and the implementation was divided into two areas, one which dealt with the development of the kernel and the management of all resources and the other which dealt with the design of the filing system and the line printer spooler. A third area is the development of the editor which was taken up separately.

In this thesis, the details of the filing system and the lineprinter spooler has been given. The second and the third chapters deal with the hardware and the software environment of the operating system. The fourth and fifth chapter give the details of the filing system and the disk management. The spooler is described in the sixth chapter. The seventh chapter gives some idea of the accounting and error handling. The same chapter elaborates on improvements possible in the design. The appendix contains the program listings.

CHAPTER 2

HARDWARE ENVIRONMENT

In this chapter a brief outline of the mini-computer TDC-316 has been given. First certain features of the hardware have been described. Later a section on the disk drive controller has been included. The last section gives a general description of the line printer controller.

2.1 THE TDC-316 COMPUTER

The TDC-316, the machine for which this operating system has been designed is a small, high speed mini-computer with a word length of 16 bits and parallel 2's complement arithmetic [TDC]:

The basic systems has 28K words of core memory expandable to 128K words with the MAP option. It has a dual bus structure - the intermixed bus and the memory bus. High speed devices like the magnetic tapes and disks can be connected as DMA devices to either bus. There is an 8 level automatic priority interrupt scheme provided with the TDC-316. DMA devices can gain control of the bus by generating non-processor request signals which have higher priority than interrupts. Nesting of interrupts is allowed.

The system has 15 addressable general purpose registers. Besides these, it has a few other special registers like the processor status word. This at any time gives the current state of the machine. The processor can run in 2 modes, the user (or unprivileged mode) and the supervisor (or privileged mode). It maintains two stacks, one for each mode of the processor. There are powerful trap features built in the system to detect illegal program specification and certain hardware conditions.

The MAP option maps out different physical spaces in memory in the user and supervisor mode. The MAP option is used to provide the mapping or relocation of the logical address space available to the programmer into a larger physical address space. In each mode 36K of virtual memory is available (28K for Instruction and 28K for Data). In addition 4K of I/O cum ROM addresses is common to all spaces.

In each area, the 32K total addressable space is further divided into 8 segments each of whose length can vary from 32 words to 4K words. Each segment must begin from a 32 word block boundary. The MAP option has hardware registers to keep the starting addresses of each segment in each mode.

The TDC-316 has all the standard peripherals of a line printer, card reader, high speed paper tape and punch and a disk unit. In the subsequent sections, we describe both the lineprinter and the disk unit in some detail.

2.2 THE DISK DRIVE CONTROLLER

The controller has a central section termed the adaptor and a section to control the disk drive peculiarities. More than one disk drive can be connected to the central section. Basically the adaptors functions are to receive, interpret and rout commands to the drive for which they are meant. It also controls data transfer by assembling and disassembling the data flow, providing intermediate buffers etc. On the completion of a job it transmits an interrupt signal to the CPU. [TDC] .

The formatter takes care of each drives peculiarities. It executes the commands passed on by the adaptor and provides the interfacing between the drive and the controller. The disk subsystem has 2 interrupt vectors, one for completion of data transfer operation and the second for non data transfer operations. These vectors are located at addresses #200 and #204 respectively.

2.2.1 The Disk Unit

We use the disk pack drive at our installation. It has an unformatted capacity of 7.25M bytes. There are 203 tracks, 10 sections per track and 10 surfaces.

There are eight registers maintained here.

- 1) Sector and Surface Register : This stores both the sector
(Address #177444) and surface number.

- 2) Attention Register (#177446) : An interrupt on account of a non data transfer operation sets a bit here which specifies the port interrupting. This is necessary since more than one drive can be connected.
- 3) Drive Status Register (#177450) : This gives the status of the drive. It can be read after a SENSE is executed. (SENSE - refer to Command Register).
- 4) Track Register (#177452) : This stores the current track number.
- 5) Command Register (#177454) : Through this various commands can be given to the disk controller. There is a SENSE command, which when set loads the current port and unit number in the DRIVE STATUS Register, which can now be read.
- 6) Controller Status Register (#177456) : This is a read only register. It is kept in the adaptor and contains status information regarding mainly data transfer operations.

- 7) Current Address Register : This keeps the memory address
(#177460) from/to which the data transfer must take place.
- 8) Word Count Register (#177462): In this must be loaded the
2's complement of the number
of words to be transferred.

2.3 LINE PRINTER CONTROLLER

The line printer provided with the TDC-316 prints 132 columns across the line at the rate of 300 lpm using a full line buffer. Characters to be printed are given to the printer serially and are stored in the line buffer of the printer.

[TDC]. Four registers are maintained here -

- 1) Line Printer Control and Status Register (LCSR) #777630
- 2) Data Buffer Register (LPBR) #777632
- 3) VFU Register (LVFR) #777634
- 4) Current Count and Status Register (CCSR) #777636

The current count register provides the line count and the character count. Line count is cleared after a page eject command. Paper movement can also be controlled by the VFU register. Characters are loaded into the printer buffer until the 'Buffer full' bit is set. Now a print command may be given through the LCSR. The characters are loaded in 6-BIT ASCII code. All lower case characters are converted to upper case and printed. Two interrupt vectors (1) error interrupt (#160) (2) ready interrupt (#164) are available.

CHAPTER 3

SOFTWARE ENVIRONMENT

The design of this operating system has been kept relatively simple. Here to each terminal is attached a job which automatically serves as the users job when he logs in. There are no back-ground jobs. [Thom 72].

In the main memory, space is allocated for the kernel programs the editor and other utilities. The rest of the core space is divided into users' work spaces, depending on the number of users, a parameter that can be specified when the system comes up. Each user has a communications area, a data area and a stack area. In the user mode of the process the MAP OPTION is set up to point to the current users work area, otherwise in the supervisor mode the MAP points to the kernel routines.

An interrupt from any peripheral device or the clock and the invocation of any trap call results in a switching from user to supervisor mode. Here the supervisor map is used. Still both modes share a common instruction segment, which is segment No.6 in the supervisor mode and segment 0 in the user mode.

The filing routines all run as an extension of the user's job but since this code is sharable, all users must have access to these routines. Therefore for each user there must be a common segment available containing the filing routines. A user can invoke the filing primitive routines via trap calls. The user's map is then changed to point to the appropriate routines and the user's PC is loaded with the start address of the invoked routine. A user in this state is then said to be in the FILING MODE. This is the 3rd mode of the processor but it is entirely simulated by software. It is useful in the sense that any disk I/O request by the user can only be made in this mode since the disk driver is hidden from the user in the filing system. Hence a security check is done at the time a request for data transfer from the disk is made by any user.

Certain data segments in the filing system are locked when being used by one job to preserve the integrity of the system. These can be thought of as resources and since all users can invoke the filing system concurrently, they must compete for the use of such resources. With each resource is associated an event number. Binary semaphores and event queues are used to synchronize the allocation and release of these resources. A user may thus be blocked on a particular event and put in the appropriate queue. When a running job releases the resource, the first job in the event queue associated with that resource is woken up [BRIN 73].

Now each invocation of a filing primitive may result in a sequence of disk I/O operations. Since all the primitives are run as an extension of the user's job, the job must be blocked at the initiation of any disk I/O and woken up when the DMA transfer has taken place. Thus here too a blocking and awakening sequence is used but there are no queues maintained.

The blocking, awakening and queue management routines are a part of the lowest layer of the kernel. For further details a companion thesis [PAND 82] may be seen.

When a user's job is blocked on initiation of disk I/O, the next user to invoke a filing primitive may need to do some data transfer from/to the disk too. Since a new I/O operation cannot be initiated till the first is completed, the user's request for disk I/O must be entered in a queue and taken up later. This queue of requests made by jobs to accomplish a disk I/O operations is maintained by the disk manager. A user's job is blocked when his request is entered in the queue. The interrupt servicing routine of the disk wakes up the last job for whom the I/O is completed and takes up the next request in the queue. This queue is thus interrupt driven.

Since all the interrupt servicing routines run in supervisor mode, the queue and its management have been made a part of the kernel which too runs in supervisor mode. Now the user enters his request via trap calls.

Here since all data transfers are to or from memory locations in user's area, we need inter-mode communication. Thus either along with a request the 18 bit actual physical memory address is loaded, or at the time of initiation of I/O a routine is called to convert the relocatable 16 bit address into an 18 bit address in user's area.

Both the card reader and the line printer spoolers on the other hand are treated as user processes in themselves. All other users communicate with them to get their files read or printed. The spooler too maintains a table of print requests. This table is accessible to all users for entry of their requests. All requests must be regarding files stored on the disk. The spooler being a user's process can invoke filing calls like OPEN and GET. Also at the initiation of any disk I/O operation it will get blocked. Like all users, it maintains a single file buffer in its area as long as file is open. The main purpose of the spooler is to get sequential blocks of the file and transfer characters line by line to a ring of buffers in the kernel. This ring of buffers then contain lines which are printed in the order in which they are filled.

Since the operating system has been designed basically to cater to the editing needs of users and to provide them adequate filing facilities and as all the users expected to use the system have a common purpose, no user's job is given priority over another. Hence the jobs are scheduled in a

round-robin manner. Also there is no swapping in the memory management. A buffer of the size of one disk sector is kept fixed for each open file in the user's area and is used for all data transfers concerning the file. Since text editing is the main purpose of the system, a file is created only as a stream of characters. A file may also be created through cards but only by a logged-in user.

Error handling is of two kinds. One due to which jobs are aborted and the other where only appropriate messages are relayed back to the user but the job continues to be in run state.

A link has been established with DEC-10 which supports several languages so that users may submit their files created on the TDC-316 for execution on the DEC-10.

CHAPTER 4

FILING SYSTEM

We first give a brief picture of the overall system as a hierarchical model. The detailed description of the design of each layer in the hierarchy is then specified. This is spread over Chapters 4 and 5.

This chapter deals with the concepts and details of the filing model which converts a user's request for some filing function to a sequence of actual disk I/O requests. The peculiarities of the disk driver and the scheduling and control of disk I/O requests is the subject matter of the 5th chapter.

4.1 HIERARCHICAL MODEL

A general description of this model has been given without making each layer specific. The attempt has been to describe the system from the outermost layer, namely that which is visible to the user and the innermost layer of actual disk I/O operations [SHAW 74].

At the outset, we shall look at the filing system from a user's view. For a user, filing facilities are available either directly by invoking monitor commands or indirectly through utilities like the editor, spooler etc.

A directory of files owned by each user is maintained. This directory may contain text files and other directories and any level of such nestings of directories may be allowed. Since the directory structure is a tree, each user's directory has a path in the tree and a user can access his files by giving a path specification. The path can be specified from the user's current directory (into which he logs-in) or from the root directory.

Each file has a descriptor which contains all relevant information about the file. Descriptors also store the disk sector addresses of the blocks allocated to the file. A file may have more than one descriptor and hence can be of any length since all its descriptors are chained together and give sequentially the addresses of the file blocks. There is a generalized protection scheme dividing users of files into (1) owner, (2) of the same project and (3) others and is used effectively to control access to files.

If a user wishes to read or write into a file, the file must be activated. This implies that the file descriptor must be read into the memory. A table of all active files is maintained. This table is searched to ensure READ-ONLY sharing of files when a file is opened (i.e. activated). Thus if a file is opened for READ-WRITE, there must be only one active copy of the file.

File blocks may be accessed by using logical block numbers starting from 0 which points to the first block.

All data transfers regarding the file take place from or to a fixed buffer in the user's memory; specified by the user when the file is opened. Thus the filing system does not itself do any buffer management. A file is always treated as a stream of characters and all transfers are in terms of file blocks of one sector size. There is no notion of records. This can be built over the system easily if required since random accessing of blocks is possible.

When the user has completed all transfer operations on a file, the file is closed i.e. deactivated and the entry in the active table is freed.

Free blocks on the disk are those which have not yet been allocated to any file. A linked list of all such blocks is maintained, and allocation/deallocation of blocks take place from the head of the list. All descriptors are kept on certain specified tracks. Their current state is maintained by a BIT MAP spread over 2 blocks in track 0. A bit when set indicates that the descriptor pointed to logically is allocated to a file, when reset it implies the descriptor is free.

Since filing routines can be invoked concurrently by all users, a queue of requests to facilitate the actual data transfer operations is maintained. This queue is interrupt driven, jobs are blocked on entering their requests in the queue and woken up after the I/O is over. This

section of the filing system i.e. the scheduling and controlling of all I/O requests is maintained in the kernel. There is a header block which keeps all the information about the disk organization; including the head of the free block chain, the addresses of the blocks containing the BIT map for the descriptors and other such specific information. This is always kept in memory when the system is initialized and written back when any modification is made, while still keeping the new copy in memory.

These in a nutshell were some of the salient features of the filing system. Detailed descriptions are given in the following sections. The appendix contains description of all relevant tables.

4.2 DETAILED DESCRIPTION OF THE SYSTEM

A file system is that software responsible for creating, destroying, organizing, reading, writing, modifying and controlling access to files and for the management of resources used by files. The filing facilities are available to the user by invoking any one of the 12 primitives:-

CREATOR, CREDIR, OPEN, CLOSE, DELETE, RENAME, PUT
GET, PUTEOF, BLKCHK, CDPCHK, SEEK (See Appendix I)

4.2.1 Directory Structure

The directory structure of the system is a tree and each directory is stored at some node in the tree. Directory

entries are branches which point to other directories or to text files which form the leaves of this tree. At the root of the tree is a master file directory owned by the system. Here nesting of directories to any level is possible hence a user may have many levels of SFD's in his area. The system deals with only 2 kinds of files - directory and text. Each file has^a main descriptor which contains relevant information about the file [RITC 78a]. Figure 4.1 depicts an example of such a directory structure.

4.2.2 User's View

A file, as mentioned earlier, is uniquely identified by its path in the directory tree structure. The user can specify the path as follows:-

Directory 1/Directory 2/Directory 3/...../Filename

Here search for the file begins from the users directory pointed to by its CDP (i.e. current directory pointer). To specify the path from the root, there must be a leading "/" (slash) in the pathname. This description is at the monitor level [RITC 78b].

To the filing system, a pointer to the pathname is passed, which itself is stored in contiguous words in the communications area of each user by the system. The format here must be as given below:-

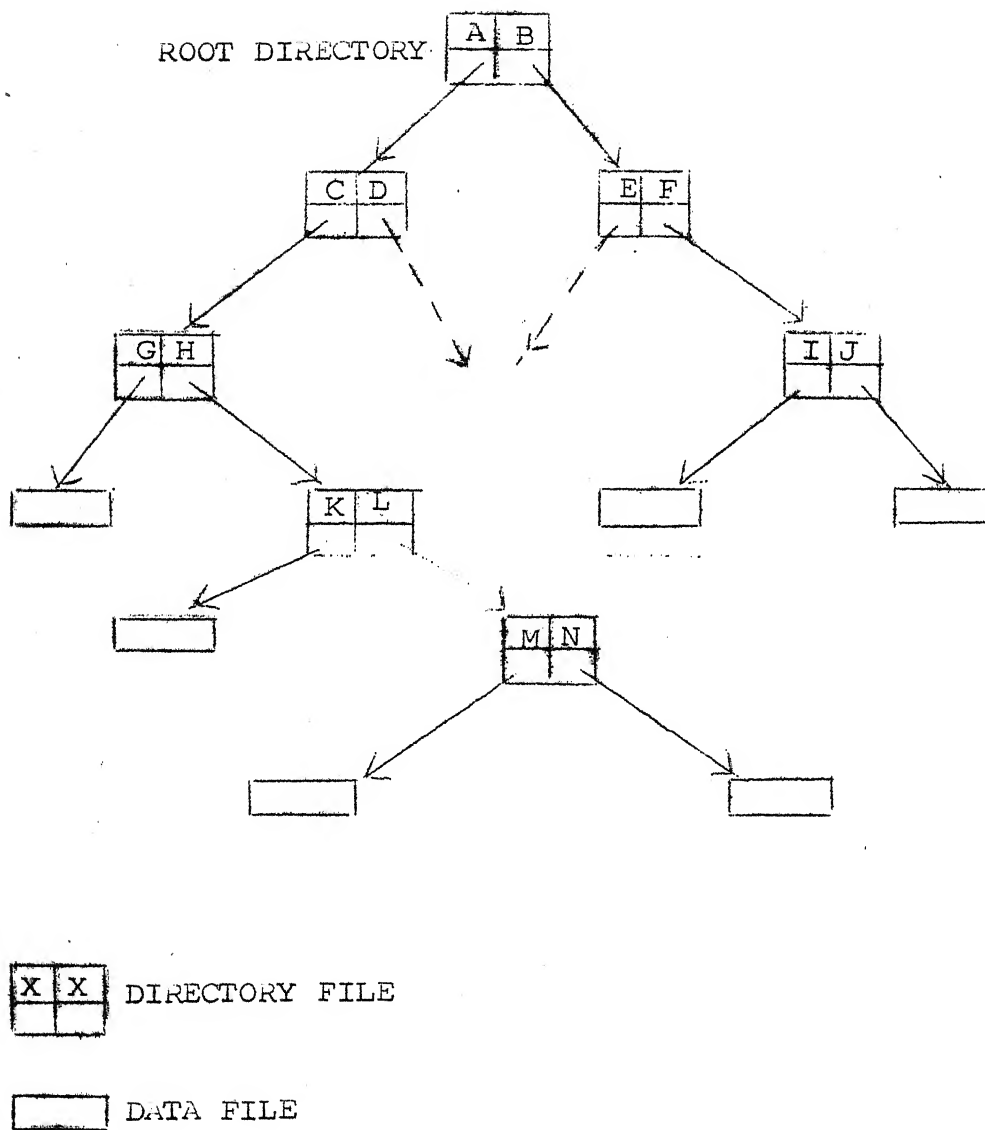


FIG. 4.1 A TREE STRUCTURED DIRECTORY

[1st word] This contains an address as given by the CDP
or -1 if search begins from the root.

[FILE NAME] [FILE NAME] [FILE NAME]....[FILE NAME] [EOLIST MARK]

Each filename is 2 words long and all filenames except the last must refer to a directory file on the path. The EOLIST MARK is a '\$' (dollar) sign placed at the end of the list.

4.2.3 Descriptors

The descriptor for a file is 64 words long (half a disk sector size). Its contents are typically:-

- 1) Filename in radix-50 code.
- 2) Owner's PPN.
- 3) Kind of file (i.e. directory or text).
- 4) Protection for the file.
- 5) Address of the directory file to which this file belongs.
- 6) Its own address
- 7) Address of the next descriptor of the file (or -1 if the file has no more descriptors).
- 8) Date of creation.
- 9) Storage map.
- 10) Size of file (i.e. number of disk blocks allocated to the file).

In the descriptor there is space for storing 54 addresses of blocks allocated to the file. If the size of a file is more than 54 blocks then another descriptor is allocated to the file and links between the 2 descriptors are maintained. This chaining of descriptors for a file is possible to any number implying thereby that there is no restriction on the size of the file [RITC 78b] .

Second and later descriptor contents are (1) address of the previous descriptor (2) address of the current descriptor (3) address of the next descriptor if any or else -1 and (4) storage map for the file.

4.2.4 CDPCHK & CREDIR

To set the current directory pointer for a user, he must specify the path of the directory. The current directory pointer or (as we shall call it from now on the CDP) essentially gives the address of descriptor of the directory file to which a user logs into.

The primitive CDPCHK is invoked for this purpose and the path of the directory from the root is traced and checked for read-only access atleast for the user. CREDIR is a primitive invoked to create a directory file. The user/system wishing to create a directory must specify its owner along with the primitive. But for text files the owner is the same as the owner of the directory to which the file belongs.

A directory file is treated differently from a text file

in that it cannot be explicitly written into. Entries are made in the file when a file is created or entries can be removed when a file is deleted. Reading of a directory file implies the looking up of filenames of files belonging to the directory. Figure 4.2 shows a typical block in a directory file.

Each entry in a directory is 3 words long. First two contain the filename of the file and the 3rd word contains the descriptor address of the file. After the last entry in the file a mark indicating the end of file is put. When a file is deleted, its entry in the directory is replaced by a -1 sign in the first word.

The root directory is stored at a fixed block and initialized when the system is set up. The address of the root directory is stored in the Header Block of the file.

4.2.5 Protection Mechanism

Users of files can be divided into 3 groups (1) owner (2) of the same project and (3) others. A hierarchical scheme is used to control the access. Each group of users are allowed either one of the following [MADN 74].

- | | |
|--|---------|
| *) No access | Value 4 |
| *) Read only access | Value 3 |
| *) Read write access | Value 2 |
| *) Read, write, delete access | Value 1 |
| *) Read, write, delete & change protection | Value 0 |

FILE	NAME 1	DESCRIPTOR ADDRESS	
FILE	NAME 2	DESCRIPTOR ADDRESS	
-1	0	0	← FREE ENTRY


FILE	NAME 3	DESCRIPTOR ADDRESS	
\$	0	0	← END OF DIRECTORY FILE
LAST TWO IN EACH	UNUSED WORDS BLOCK		

FIG. 4.2 A TYPICAL DIRECTORY BLOCK

Here a 3 digit number with each digit in the range 0 to 4 is used to keep a check on the kind of access made by any user on the file. When a user wishes to access a file, he must be allowed to look up directories on the path of the file from the root. Hence he must have atleast read only access on all directories on the path. If he wishes to create, delete or rename a file then the user must have write access on the directory to which the file belongs, and he must have the specified access on the file itself. The only variation in this rule is in the case of owners who are always allowed to change their protection whatever be the protection specified earlier.

Thus a general protection mechanism is used to control access to all files. This also imposes an order in which directories are allocated to users. Since users must have read access on all directories on the path, hence the path must have directories owned by users who are willing to let other users in the path look into their area. This is general, in the sense that a user may create a file in another users directory provided he has write access on the directory and read access on the path; but for deleting another user's file, the first user must also have delete access on the file.

4.2.6 Activation of Files

When a user wishes to access a file for reading, writing or modifying the file must first be activated. That

is, the main descriptor of the file must be read into the memory. To facilitate this a table of all active files is maintained. Each entry contains a pointer to the descriptor of the file in memory. Such a table is called the ACTIVE TABLE (See Appendix II for a detailed description of this table).

A file is activated by invoking any one of the two calls, CREATOR and OPEN. CREATOR creates the file entry in the directory, allocates and initializes a descriptor for the file and then activates the file exactly like in OPEN. This primitive is used to create only text files [SHAW 74., SHAW 79].

First the block address of the descriptor of the file is obtained (both in CREATOR and OPEN) after checking for atleast read only access on all directories on the path and in the case of CREATOR, for write access on last directory. Hence the user's PPN must be supplied with the primitive.

A file may be opened in either one of the two states, READ and READ-WRITE. Since read-only sharing of files is allowed, before a file is opened the ACTIVE TABLE is searched and the descriptor address of the file is matched since this address uniquely identifies a file. If the file is to be opened for READ-WRITE and it has already been opened earlier, an error message is returned. Again if a file is to be opened for READ and it has been opened for READ-WRITE earlier, the error message is returned and the job is aborted in both cases.

A file to be created is always opened for READ-WRITE and is of size 0. Whenever a new block is allocated to the file, the size is increased by one.

To open a file for READ-WRITE the user must have write access for the file and to open for READ the user should have read access in the file.

With each file certain other status information like the user's job no., the current size of the file, the current descriptor number for each file is maintained. Other pointers and flags like the current block pointer etc. too are kept. Their use shall be clear from the next section.

Channel numbers starting from zero are used to index the ACTIVE table. These are also returned to the user invoking CREATOR or OPEN. Now an active file can be accessed by the job which opened it by just referring to the channel no.

Directory files can never be opened for writing and hence CREDIR does not return any channel number to the user.

4.2.7 Reading, Writing and Modifying

For each open file in the Active Table, is maintained a current block pointer. All reads and writes of the file are with reference to this pointer. A file is treated only as a stream of characters and users may get (i.e. read) blocks of the file into their memory area and also put (i.e. write) blocks of the file from their memory area. The current block number is initially zero and points to the first address

in the storage map of the file descriptor. Thus specific blocks of the file can be accessed by their logical block numbers from the start of the file. When a GET or a PUT primitive is invoked, the logical block number is obtained from the current block pointer. The address of the specified block can now be read from the descriptor. This address is used to initiate the data transfer. After the operation is over, the current pointer is incremented and the new value is returned to the user. The next GET or PUT will result in a data transfer from the block currently pointed.

A user can set his pointer to any specific block by using a SEEK command. A user may also check the current value of the pointer by a BLKCHK command [RITC 78a].

Another primitive available here for ACTIVE files is the PUTEOF. This puts an end of file mark in place of the address pointed to by the current block pointer in the descriptor. The remaining blocks allocated after this mark are then released.

The pascal primitives RESET and REWRITE can be implemented easily. For RESET a SEEK (0) will reset the pointer to the beginning of the file and for REWRITE a SEEK (0) and a PUTEOF after it will delete all the blocks of the file and reset the pointer to the beginning, while still keeping the file ACTIVE.

For all data transfers regarding ACTIVE files the user

must specify a memory buffer in his area, with the OPEN or CREATOR primitive. The address of this buffer is stored in the ACTIVETABLE entry for his file.

4.2.8 Deactivation of Files

A file may be deactivated, essentially made inaccessible to the user for reading or writing purposes by a CLOSE command. This primitive frees the ACTIVE TABLE entry for the file SHAW 79, [RITC 78a].

When a user changes the size of the file, i.e. either by specifying a PUTEOF which frees some file blocks or by specifying PUTS which result in the allocation of a new block, a modification must be made in the size of file field in the first descriptor. This is done when a file is CLOSED.

The current size of the file is always maintained in the ACTIVE TABLE. A bit to indicate if the size has changed from its original value, is also kept. If this bit is set, the first descriptor of the file is read into the memory, the size of the file is modified in the descriptor and then it is written back.

Another flag to indicate if the current descriptor is modified is also stored. This is to ensure the writing back of the current descriptor (if modified) before another descriptor is read.

Each descriptor of a file is linked to the preceding and succeeding descriptors. Also the address of the first descriptor is kept in the ACTIVE TABLE entry for the file.

4.2.9 DELETE & RENAME

These are two more primitives whose use so far has not been mentioned. A user may invoke DELETE if he wishes to remove a file from some directory. To delete a file, the user must have read access on all directories on the path from the root, write access on the last directory and delete access on the file itself.

A directory file can only be deleted when empty. If a file is ACTIVE when a DELETE primitive for the file is invoked, then the directory entry of the file is removed while still keeping the blocks of the file. The file is now in the DELETE state. The last CLOSE command for the file then frees the blocks and descriptors of the file. No other user can OPEN the file after the entry is deleted from the directory.

The RENAME primitive is invoked if a user wishes to change the name or the protection rights of the file. In both cases the user should have write access on the directory of the file. For changing only the name, the user must have "delete" access on the file itself and for changing the protection the user should have "change protection"

access on the file [DEC] . With this section we conclude the description of all the primitives.

4.2.10 Free Storage Management

In the preceding sections, we have come across two significant facts. One where a PUT primitive may allocate a new block to a file and second, a PUTEOF may delete some blocks from a file. Exactly how is this allocation and deallocation of blocks done ?

In our filing system all the free blocks are linked together in a single chain. The head of this chain is stored in the the first word of the HEADER block. Since a copy of the HEADER block is always kept in memory, a number of disk I/O operations are reduced by reading the address of the next free block directly from the memory.

4.2.11 Descriptor Management

All descriptors are stored in the last 21 tracks (track numbers 179 to 199) on the disk. Their allocation and deallocation is facilitated by a BIT MAP which is kept on block numbers 1 and 2 on track 0. Each descriptor occupies only half a sector. Thus each word of 16 bits in the BIT MAP gives the current status of 16 descriptors in contiguous 8 blocks. If a bit is set, then the descriptor pointed to logically by the position of the bit in the map,

is allocated already to a file. Thus a free descriptor is indicated by a 0 value in the logical bit position for the descriptor.

4.2.12 Addressing Scheme

We have always talked in terms of logical block addresses of the disk sectors. Hence a mapping must be made from the logical address to the actual track, sector and surface address. In this section, the scheme used to form the logical addresses is described.

Each address is one word long. The first seven bits are used to store the logical block number in a cylinder. This number can vary from 0 to 99 since a cylinder has a hundred sectors. The next eight bits store the cylinder number. Since there are 203 tracks, cylinder numbers can vary from 0 to 202. The fifteenth bit (i.e. the most significant bit) is used to locate a descriptor within a block. Since two descriptors are packed into a sector, this bit when set indicates that the specified descriptor is in the latter half of the sector.

Each cylinder has 10 surfaces and 10 sectors in each surface. The block address in the cylinder modulo 10 gives the sector number and the address divided by 10 gives the surface number.

4.2.13 HEADER & ROOT

Both of these are of a half sectorsize and together occupy block zero on track zero. Thus they are both packed into the same sector [DEC]. The Header contains the details of the various disk blocks on the file. Its contents are typically:-

- (1) Head of the free blocks chain
- (2) Address of the first block storing the BIT MAP for descriptors
- (3) Address of the second block
- (4) Starting cylinder number for storing descriptors
- (5) Last cylinder number used for storing descriptors
- (6) Numbers of blocks used for storing descriptors
- (7) Starting block address for the Accounting file
- (8) Last block address for the Accounting file
- (9) Number of blocks used by the Accounting file.

The root is the descriptor of the master file directory of the system. It is created when the system is set up. The owner of this file is the system which has a privileged PPN.

4.2.14 The PPN Scheme

A PPN is one word long. It is divided into 2 fields which together uniquely identify the user. Each field occupies one byte. The first field (i.e. higher byte of the

word) indicates the user's project number and the second field (i.e. the lower byte) indicates the user's programmer number.

Octal numbers are to be used in both cases. Hence each field can range between #0 and #377 [DEC].

4.2.15 System Initialization

To set up the system appropriately an initializing procedure must be followed for the filing system. First the free blocks of the disk must be linked, then the Header, the Root and all the tables must be initialized. The Disk subsystem too must be initialized here.

If and when a crash occurs or the system is shut-down and restarted, the above initializing procedure need not be followed. Here, only the tables and the disk subsystem needs to be re-initialized.

4.2.16 Synchronization

As explained earlier each primitive runs as a part of the user's job. Now when any disk I/O is initiated by the primitive, the user's job is blocked. Another user may now invoke a filing primitive.

Thus all primitives can be invoked concurrently and each may be in a certain stage of execution when blocked. This imposes some restrictions on the filing system to

maintain its integrity. An example of this is given below.

When a free block is allocated or deallocated the contents of the Header block are modified. A user's job may be blocked when the modification is not complete since getting or putting a free block results in more than one disk I/O operations. Another user thus, must not be allowed to access the Header while this modification is going on. Hence with any getting or putting of a block an event number is associated. Binary semaphores are used to lock out this resource and all processes desiring this resource are entered in an event queue [BRIN 73] for the resource.

The same principle is applied in the case of the free descriptor management to preserve the integrity of the BIT MAP. Details of this scheme may be seen in [PAND 82].

There are certain buffers maintained and some global words which reflect the state of the currently invoked request in the case of some filing primitives.

These buffers are mainly used to search a directory for a specific file entry, while the trace of the file is going on in the directory structure.

Here again an event number is associated with this operations and all primitives invoking this operation have to wait on this event. These primitives are OPEN, DELETE, CREATOR, CREDIR, CDPCHK and RENAME.

Thus only one such primitive can be in operation at a time. The other primitives GET, PUT, SEEK, BLKCHK, PUTEOF, CLOSE can run concurrently with each other and with the above six.

4.2.17 Entering of Requests

A filing primitive may often in its execution require to do some disk data transfer operation.

For this purpose, the primitive makes a request via a trap call to the disk-manager. This request is entered into a queue maintained by the disk-manager.

Each request can specify any one of the four functions, (1) Reading of a block, (2) Writing of a block, (3) Reading of a descriptor and (4) Writing of a descriptor. With each request, the block address, the memory location from or to which the data transfer takes place and the job number must be specified. After entering the request the user's job is blocked. It is awakened when the data transfer is over.

CHAPTER 5

DISK MANAGEMENT

This chapter gives a general description of the disk-manager. It also contains a brief account of the disk interrupt servicing routines.

5.1 INTRODUCTION

The disk manager, as has been mentioned earlier, is a part of the kernel. It is designed to control and complete any data transfer operations from or to the disk. Thus all disk accesses must go through the disk-manager.

Mostly disk requests are expected to come from filing system routines. But the system too can ask for any disk service from the disk-manger. The manager maintains a queue of all disk requests and initiates the next one when the current request is over. It also does the useful task of waking up the system/user's job which was blocked for the current I/O.

In the last chapter, there was an account of how requests are entered in the queue by the filing routines. For any other system utility or program, a similar call would be sufficient to invoke the disk-manager. Hence the

entering of requests in the queue is not dealt with in this chapter. We start here from how the I/O operations are actually carried out.

5.2 QUEUE OF REQUESTS

Each request entered in the queue must specify the following:-

- 1) The job number of the invoking user.
- 2) The operation desired: This is taken up in more detail in the next section.
- 3) Memory location: This is the address in memory from or to which the operation is desired. This in most cases must be an 18 bit address, 2 bits of which are stored with the earlier field, as this field is 1 word long only.
- 4) Disk block specification: Here a block address of the disk sector from or to which I/O must take place is given. The format for this address has been described earlier in Section 4.2.12.

The table maintains two pointers, an IN and OUT pointer. IN points to the next free space available in the table for entering a new request. Thus IN serves as an index to the table when a request is entered. OUT points to the request whose disk I/O operation is going on currently.

IN and OUT are both initialized at zero. A 'PENDING' flag indicates if a disk operation is currently being carried out. If IN is equal to OUT and the PENDING flag is off then a queue empty condition is indicated, otherwise if the PENDING flag is on and IN equals OUT, then a queue full condition is indicated. In the latter case, a user trying to enter a request in the queue is returned an error message.

If the queue is empty then after the first request is entered, the I/O operation is initiated. An interrupt occurs when the operation is complete. The interrupt servicing routine then checks to see if request has been entered meanwhile in the queue, in which case it initiates the next operation. The queue in this sense is largely interrupt driven. On the completion of each request the appropriate job is signalled.

5.3 OPERATIONS

When a request is entered in the queue, any one of the four operations can initially be specified. They are:-

- 1) Operation 0 - read a block
- 2) Operation 1 - write a block
- 3) Operation 2 - read a descriptor
- 4) Operation 3 - write a descriptor

The first two operations are simply the reading of and writing into a disk sector. Here an 18 bit address

must be specified for the data transfer operation to take place with the MAP option in memory. The last two operations are necessary as a descriptor is only of half a sector size. Thus reading and writing of a descriptor from the user's point of view is the transferring of only half a sector.

Hence in both cases, some housekeeping must take place either before or after the operation. Also each whole operation must be indivisible. This implies that no other disk I/O request should be initiated while either of the two operations is going on. Here by the term 'operation' we mean the disk access along with the housekeeping needed. A scheme has been devised to ensure that this integrity is maintained. We shall take up each case separately.

5.3.1 Reading of a Descriptor

This is operation '2' as specified by the user. After the first disk access (i.e. reading of the sector specified by the descriptor's address into a buffer local to disk manager) is initiated, the operation is changed to '4'. The OUT pointer in the table is not incremented. Here the user need only specify a 16 bit relocatable address in his area since the 18 bit address of the local buffer is entered into the controller registers.

When the interrupt servicing starts, in general, the table being interrupt driven, the next request is taken up

and initiated. The next request is always pointed to by OUT. Here the next request is the same one with the operation changed to '4'. So now operation '4' is carried out. That is, the descriptor is transferred to the user's area. After this, the job which had made the request is informed of its completion. The OUT pointer is incremented and the next request is taken up.

5.3.2 Writing of a Descriptor

This is operation '3' as specified by the user. After the first disk I/O operation (i.e. reading of the section into a local buffer) is initiated the operation is changed to '5'. Again the OUT pointer is not incremented.

So the next request to be taken up when this read operation is over, is the same one with operation '5'. Operation '5' transfers the specified descriptor from the users area to the appropriate half of the sector read into the local buffer. The buffer is then written back into the sector. After this, the job is informed of the completion of this request and the next one is taken up.

This, briefly describes the two operations of reading and writing a descriptor. In the case of 'write' too, the user must specify only a 16 bit relocatable address in his area.

5.4 INITIATION OF I/O AND INTERRUPT SERVICING

To initiate the disk I/O operation the appropriate disk controller registers must be filled and the command to be given, entered in the controller.

For this a current request table is maintained which keeps track of the operation on at present. In this table only a 'read sector' or a 'write sector' operation can be specified. The purpose of this table is mainly to allow retries (upto a maximum of five tries can take place) of the same operation in case an error occurs.

After the operation is completed, the interrupt servicing starts. The interrupt servicing routine drives the queue. At each interrupt it checks for the correct completion of the operation. In cases of errors, it initiates the next try and keeps a count of the number of retries. Otherwise it wakes up the job blocked on this disk I/O request and takes up the next request in the queue. In the case of operations 2 and 3, the waking up of the job is postponed to until 4 and 5 respectively are over. If there is no further request in the queue, it resets the PENDING flag and returns.

Here non-data transfers are not initiated separately, hence the only case in which a non-data interrupt servicing is needed, is if an error occurs in the seeking or searching

of a track and sector. This interrupt routine then checks for the error, retries the operation if possible and returns. As the operation is always a data transfer operation the next interrupt (if the operation completes successfully) should be serviced by the data transfer servicing routine.

5.5 CONCLUSION

The model for the disk-manager has been described in this chapter. A necessary optimization could be made in the handling of the driver (i.e. initiation of operation and interrupt servicing) by checking and handling each kind of error appropriately. At present, all errors are treated in a similar fashion and after 5 retries, the request is aborted.

CHAPTER 6

THE LINE PRINTER SPOOLER

This chapter deals with the spooler to print files. It first gives^a general description of the spooler process and later of the line printer driver.

6.1 THE SPOOLER PROCESS

The spooler runs as a user process. Other user's communicate with it to get their files printed. This communication is through a call 'ENTERSPOOL' which enters the user's request into a queue maintained by the spooler [SHAW 79].

The spooler first activates the next file in the output spool list by invoking the file primitive OPEN. The file header record "the Banner", which is essentially the user's PPN, is printed to help users to identify their files. Then blocks of the file are read successively (by invoking the filing primitive GET), into a buffer kept in the spooler area. After a file block is fetched, it is transferred to a ring of buffers of one line size each maintained by the line printer driver. Only after all the characters in the block are transferred is the next GET

invoked.

The line printer driver then initiates the printing of each input full buffer. The spooler process may be blocked on two events. (1) When it waits for a disk operation to be completed. This occurs when it invokes any one of the file primitives OPEN and GET or even CLOSE. After all the blocks of the file have been transferred CLOSE must be invoked to deactivate the file. (2) When it waits for an empty one line buffer. This may occur if the buffers are filled up faster than they are emptied (i.e. printed).

6.1.1 The Spool Table

The spooler maintains a queue of all requests for the printing of files. With each request the user must specify (1) his job number (2) pathname of the file (3) the user's PPN (4) page limit for the file. A sequence number is added by the spooler. This number uniquely identifies a request in the queue and may be used later to kill or modify a request or see its status. Since only three words are available for storing pathnames, the user must specify the path from the directory which contains the relevant file. There is provision for only one file entry per request. SIN is used to enter the next request and SOUT to remove the next request from the queue for printing [MADN 74].

6.2 LINE PRINTER DRIVER

The line printer maintains a ring of 15 one line buffers. Each line can have a maximum of 132 characters. The buffer size has been kept at 134 characters to store end of line and end of file characters too in the buffer. Buffers are of 3 kinds:- IN, OUT and EM (i.e. input full buffers, output buffers and empty buffers respectively). There is only one output buffer at any time which is currently being printed. [SHAW 74].

A single chain is used to link all input buffers together and another single chain to link all empty buffers. Pointers to the head and tail of each chain are maintained. Each buffer is referenced by a number.

Two routines ADDBUF and TAKEBUF are called to add or remove a buffer from a chain. Semaphores are used to keep a count of the number of buffers in each chain and synchronize the allocation and deallocation of buffers.

The primitive SENDLINE is invoked by the spooler to get an empty buffer and fill in a line and add it to the input buffers chain. PUTLINE is called by the driver to initiate the printing of the line contained in the first buffer in the input chain. A count of the number of lines printed per page is maintained to help control the movement of paper.

This in brief gives a general description of the spooler. The programs have been listed in the appendix. Due to a shortage of time, the printing of the banner has been restricted to the PPN only. An improvement can be made by including the filename and date too. Also the printing of banners in giant letters could be incorporated.

CHAPTER 7

CONCLUSION

In this chapter a brief description of the accounting procedure is given. There is a section on the error handling. The last section suggests improvements that could have been made and gives an idea of the further work possible in the system.

7.1 ACCOUNTING

This has not been implemented. As such this section serves as a guide to the accounting procedure possible.

For each user an account record is maintained which contains:- (1) His PPN (2) PASSWORD (3) Disk Log-Out Quota, (4) Connect Time and (5) Disk Quota Used. All the accounting records are stored on specified disk blocks. Space has been allocated for them from blocks 4 to 99 on cylinder 0. This information is stored in the Header.

The accounting blocks do not constitute a file and thus do not have a descriptor. They are accessed by invoking any one of the four primitives:-

- (1) INSERT (PPN)
- (2) REMOVE (PPN)

(3) MODIFY (PPN, FIELD, NEWVALUE)

(4) EXAMINE (PPN)

INSERT adds a new PPN and the corresponding record into the specified block. REMOVE deletes a PPN with its record. MODIFY will change the value of the FIELD specified to the NEWVALUE. Examine returns the value of the FIELD.

Given the PPN a hashing function can be used to identify the required disk blocks amongst those allocated for the accounting records. A hashing function has been suggested below.

The PPN has 2 fields, a project number and a programmer^{no.} each varying between #0 and #377. The project field modulo 64 can be used to refer to the first 64 blocks. The PPN can now be searched for in the block obtained. Overflows from the blocks are stored in the last 32 blocks.

The primitive MODIFY will have to be incorporated into two filing routines CLOSE and DELETE to maintain the integrity of the number of disk blocks owned by the PPN.

7.2 ERROR HANDLING

Error handling is of two kinds. In one case an error value is returned to the routine invoking the filing primitive and in the other case the user's job is aborted,

his stack reinitialized and an error message displayed.

The error handling is not complete. Appropriate messages have to be displayed. Appendix III contains a list of error messages. All error values begin from the number #077000. The values associated with the messages are added to this number.

7.3 REMARKS

The filing system and the spooler have been described in the earlier chapters. Regarding both of them, some remarks have been included here. These include both suggested improvements in the system and an account of its current status.

- (1) Although all the filing routines have been implemented, to make them usable, a number of system commands must be written. Examples of these are LOGIN, PRINT, TYPE, COPY etc.
- (2) Another incomplete portion is the printing of suitable BANNERS in giant letters by the spooler. Currently only the user's PPN is printed.
- (3) One of the factors which increased the complexity of the system was the execution of the filing primitives in user mode and the disk-manager in

the supervisor mode. Inter-mode communication of buffer addresses had to be incorporated.

(4) Another significant point is the half a sector **size descriptor**. Possibly descriptors occupying a full block each would have been less cumbersome.

q (5) Buffer management is not a part of the system.

A user can also specify only one buffer for each open file. This is restrictive since 2 blocks of the file cannot be in memory at the same time. Also routines to get and put characters and other status information for the buffer must be kept by each user. A common buffer pool could have localized this management.

The design, otherwise is extremely simple and general in nature to ensure that modifications can easily be made. Program listings are contained in the Appendix IV.

Acc. No. 1
CENTRAL LIBRARY

REFERENCES

1. [BRIN 73] Brinch Hansen, P., "OPERATING SYSTEM PRINCIPLES", Chapter 3, 1973, Prentice Hall, Inc.
2. [MADN 74] Madnick and Donovan, "OPERATING SYSTEMS", Chapter 5 and 6, 1974, McGraw-Hill Kogakusha Ltd.
3. [PANO 82] Pandya, P.K., "A TIME-SHARING SYSTEM FOR TDC-316", M.Tech. Thesis, Computer Science Programme, Indian Institute of Technology, Kanpur, 1982.
4. [RITC 78a] Ritchie and Thompson, "UNIX TIME-SHARING SYSTEM", The Bell Systems Technical Journal, Vol. 57, No. 6, Part 2, 1978, pp 1905-1929.
5. [RITC 78b] Ritchie, D.M., "UNIX TIME-SHARING SYSTEM: A RETROSPECTIVE", The Bell Systems Technical Journal, Vol. 57, No. 6, Part 2, 1978, pp 1947-1967.
6. [SHAW 74] Shaw, A.C., "THE LOGICAL DESIGN OF OPERATING SYSTEMS", Chapters 3 and 9, Prentice-Hall, Inc.
7. [SHAW 79] Shaw, A.C., Design of a Single Language Multiprogrammer System in "PRINCIPLES OF SOFTWARE ENGINEERING & DESIGN" by Zelkowitz, Shaw and Gannon, Chapter 4, 1979, Prentice-Hall, Inc.
8. [THOM 72] Thomas J.R., The Structure of a Time-Sharing System in "OPERATING SYSTEM TECHNIQUES", pp 351-370, 1972, Ed. Hoare and Perrott, Academic Press.
9. [WILS 76] Wilson, R., The Titan Supervisor, in "STUDIES IN OPERATING SYSTEMS", Chapter 4, 1976, McKeag and Wilson, Academic Press.
10. [TDC] "SYSTEM MANUAL TDC-316", Vol. I, II and III, Electronics Corporation of India Limited, Hyderabad.
11. [DEC] "DEC-10 MANUAL, MONITOR STRUCTURE", pp 44-55, Digital Equipment Corporation.

APPENDIX I

VALUE RETURNED TO THE USER		LIST OF PRIMITIVES
1. CHANNEL NUMBER	=	CREATOR(JOBNO,PPN,PATHNAME, PROTECTION,BUFFER)
2. CHANNEL NUMBER	=	OPEN(JOBNO,PPN,PATHNAME, FUNCTION,BUFFER)
3. MESSAGE OF SUCCESSFUL COMPLETION	=	CREDIR(JOBNO,PPN,PATHNAME, PROTECTION,OWNER)
4. DIRECTORY ADDRESS	=	CDPCHK(JOBNO,PPN,PATHNAME)
5. CURRENT VALUE OF THE BLOCK POINTER	=	GET(JOBNO,CHANNEL NUMBER)
6. CURRENT VALUE OF THE BLOCK POINTER	=	FUT(JOBNO,CHANNEL NUMBER)
7. CURRENT VALUE OF THE BLOCK POINTER	=	SEEK(JOBNO,CHANNEL NUMBER, BLOCK NUMBER)
8. CURRENT VALUE OF THE BLOCK POINTER	=	BLKCHK(JOBNO,CHANNEL NUMBER)
9. NUMBER OF BLOCKS FREED	=	PUTEOF(JOBNO,CHANNEL NUMBER)
10. MESSAGE OF SUCCESSFUL COMPLETION	=	CLOSE(JOBNO,CHANNEL NUMBER)
11. NUMBER OF BLOCKS FREED	=	DELETE(JOBNO,PPN,PATHNAME)
12. MESSAGE OF SUCCESSFUL COMPLETION	=	RENAME(JOBNO,PPN,PATHNAME, NEW PROTECTION)

NOTE: IN THE CASE OF RENAME , THE NEW FILE NAME IS SPECIFIED
IN THE PATHNAME AFTER THE END OF LIST MARKER .

APPENDIX II

ACTIVE TABLE

Each table entry consists of the following fields:-

FIELDNAME	POSITION
-----	-----
1. Job Number	First Word
2. Current Block Pointer	Second Word
3. Status Word	Third Word
4. File opened for READ or READ-WRITE	Bit 0 in Status Word
5. Current descriptor modified	Bit 2 in Status Word
6. First descriptor modified	Bit 1 in Status Word
7. Current descriptor number	Bits 3 to 7 in Status Word
8. File is in DELETE state	Bits 8 to 13 in Status Word
9. Memory extension bits of the address of the buffer used by the file.	Bits 14 & 15 in Status Word
10. Size of file	Fourth Word
11. The first descriptor's address for the file	Fifth Word
12. Address of buffer to be for the file.	Sixth Word

APPENDIX III

ERROR MESSAGES

ERROR NO.	MESSAGE
-----	-----
0	ERROR ON DISK ACCESS DATA TRANSFER
1	ILLEGAL FILE SPECIFICATION
2	FILE DOES NOT EXIST
3	FILE ALREADY EXISTS
4	FILE IS NOT A DIRECTORY FILE
5	ILLEGAL WRITE ACCESS ON DIRECTORY OPENING
6	FILE OPENED EARLIER
7	FILE OPENED FOR WRITE EARLIER
8	PROTECTION FAILURE
9	ACTIVE FILES CAPACITY EXCEEDED
10	ILLEGAL CHANNEL SPECIFICATION
11	WRITE ACCESS INVALID ON A FILE
12	ERROR!! ATTEMPT TO READ BEYOND EOF
13	DRIVE FAULT
14	NO FREE BLOCK AVAILABLE
15	NO FREE DESCRIPTOR AVAILABLE
16	DIRECTORY FILE TO BE DELETED ONLY WHEN EMPTY
17	DISK REQUEST CAPACITY EXCEEDED

% LIST OF ORDINARY MESSAGES %

NO.	MESSAGE
-1	END OF FILE REACHED


```

00530 global PATH ,
00540 ADDRESS ,
00550 CHANNEL ,
00560 STORE ,
00570 INDEX ,
00580 FERROR ,
00590 NJOB ;
00600
00610 ! stores pointer to pathname
00620 ! stores directory's address
00630 ! stores current channel number, allocated entry
00640 ! stores address of block containing file
00650 ! stores pointer to storage map of file
00660 ! stores error value when routine exits
00670 ! stores current job's number
00680
00690 external ENTERREQUEST; ! this is called to enter request in disk queue
00700
00710 MACRO OUTERR(NUM)=
00720 ( FERROR = (NUM + ERPVAL) ; RETURN -1)$;
00730
00740 GLOBAL ROUTINE FILINIT=
00750 BEGIN % initializes all filing global variables %
00760
00770 ADDRESS = PATH = INDEX = 0;
00780 CHANNEL = STORE = FERROR = NJOB = 0;
00790 INCR I FROM 0 TO 127 DO ( LINEAR(I) = 0 );
00800 INCR I FROM 0 TO 63 DO ( TRACEBUF(I) = 0 );
00810 END;
00820
00830 GLOBAL ROUTINE DISKIO(FUNC,DSKADDR,MEMLOCN)=
00840 begin
00850 ! this routine enters request for diskio %
00860
00870 Local TASK,NEWLOCN;
00880
00890 TASK<opr> = , FUNC;
00900 if .FUNC led i then
00910 begin NEWLOCN = .MEMLOCN;
00920 end;
00930 NEWLOCN = .MEMLOCN;
00940 ENTERREQUEST(.NJOB, .TASK, .DSKADDR, .NEWLOCN);
00950 end;
00960
00970 % MODE OF ACCESS OF DECRYPTOR FIELDS: [CHANNEL, <fieldname> ]
00980 % MODE OF ACCESS OF FIELDS IN ACTIVE TABLE: [ CHANNEL, <fieldname> ]
00990
01000 end ELUDOM

```

```

% author : AARTI KUMAR
% date of creation : 5 MAY, 1982
%
```

```

=====
JOBNO      FILENAME  CURRBLK  FILSTAT  SIZE  PARENT  PREVEDESC  NEXTEDESC  OWNER  DATE  STORAGE  BUFADDR  opn  descno  descbits  extension  currm  currmmod  writbit  winding  protection  latter  MAXVAL  MAXHEAT  ROOT  ERRVAL  MAXERR  NUM  INDT  POLIST
=====
0$, 0$, 1$, 1$, 2$, 3$, 4$, 5$, 6$, 7$, 8$, 9$, 10$, 11$, 12$, 13$, 14$, 15$, 16$, 17$, 18$, 19$, 20$, 21$, 22$, 23$, 24$, 25$, 26$, 27$, 28$, 29$, 30$, 31$, 32$, 33$, 34$, 35$, 36$, 37$, 38$, 39$, 40$, 41$, 42$, 43$, 44$, 45$, 46$, 47$, 48$, 49$, 50$, 51$, 52$, 53$, 54$, 55$, 56$, 57$, 58$, 59$, 60$, 61$, 62$, 63$, 64$, 65$, 66$, 67$, 68$, 69$, 70$, 71$, 72$, 73$, 74$, 75$, 76$, 77$, 78$, 79$, 80$, 81$, 82$, 83$, 84$, 85$, 86$, 87$, 88$, 89$, 90$, 91$, 92$, 93$, 94$, 95$, 96$, 97$, 98$, 99$, 100$, 101$, 102$, 103$, 104$, 105$, 106$, 107$, 108$, 109$, 110$, 111$, 112$, 113$, 114$, 115$, 116$, 117$, 118$, 119$, 120$, 121$, 122$, 123$, 124$, 125$, 126$, 127$, 128$, 129$, 130$, 131$, 132$, 133$, 134$, 135$, 136$, 137$, 138$, 139$, 140$, 141$, 142$, 143$, 144$, 145$, 146$, 147$, 148$, 149$, 150$, 151$, 152$, 153$, 154$, 155$, 156$, 157$, 158$, 159$, 160$, 161$, 162$, 163$, 164$, 165$, 166$, 167$, 168$, 169$, 170$, 171$, 172$, 173$, 174$, 175$, 176$, 177$, 178$, 179$, 180$, 181$, 182$, 183$, 184$, 185$, 186$, 187$, 188$, 189$, 190$, 191$, 192$, 193$, 194$, 195$, 196$, 197$, 198$, 199$, 200$, 201$, 202$, 203$, 204$, 205$, 206$, 207$, 208$, 209$, 210$, 211$, 212$, 213$, 214$, 215$, 216$, 217$, 218$, 219$, 220$, 221$, 222$, 223$, 224$, 225$, 226$, 227$, 228$, 229$, 230$, 231$, 232$, 233$, 234$, 235$, 236$, 237$, 238$, 239$, 240$, 241$, 242$, 243$, 244$, 245$, 246$, 247$, 248$, 249$, 250$, 251$, 252$, 253$, 254$, 255$, 256$, 257$, 258$, 259$, 260$, 261$, 262$, 263$, 264$, 265$, 266$, 267$, 268$, 269$, 270$, 271$, 272$, 273$, 274$, 275$, 276$, 277$, 278$, 279$, 280$, 281$, 282$, 283$, 284$, 285$, 286$, 287$, 288$, 289$, 290$, 291$, 292$, 293$, 294$, 295$, 296$, 297$, 298$, 299$, 300$, 301$, 302$, 303$, 304$, 305$, 306$, 307$, 308$, 309$, 310$, 311$, 312$, 313$, 314$, 315$, 316$, 317$, 318$, 319$, 320$, 321$, 322$, 323$, 324$, 325$, 326$, 327$, 328$, 329$, 330$, 331$, 332$, 333$, 334$, 335$, 336$, 337$, 338$, 339$, 340$, 341$, 342$, 343$, 344$, 345$, 346$, 347$, 348$, 349$, 350$, 351$, 352$, 353$, 354$, 355$, 356$, 357$, 358$, 359$, 360$, 361$, 362$, 363$, 364$, 365$, 366$, 367$, 368$, 369$, 370$, 371$, 372$, 373$, 374$, 375$, 376$, 377$, 378$, 379$, 380$, 381$, 382$, 383$, 384$, 385$, 386$, 387$, 388$, 389$, 390$, 391$, 392$, 393$, 394$, 395$, 396$, 397$, 398$, 399$, 400$, 401$, 402$, 403$, 404$, 405$, 406$, 407$, 408$, 409$, 410$, 411$, 412$, 413$, 414$, 415$, 416$, 417$, 418$, 419$, 420$, 421$, 422$, 423$, 424$, 425$, 426$, 427$, 428$, 429$, 430$, 431$, 432$, 433$, 434$, 435$, 436$, 437$, 438$, 439$, 440$, 441$, 442$, 443$, 444$, 445$, 446$, 447$, 448$, 449$, 450$, 451$, 452$, 453$, 454$, 455$, 456$, 457$, 458$, 459$, 460$, 461$, 462$, 463$, 464$, 465$, 466$, 467$, 468$, 469$, 470$, 471$, 472$, 473$, 474$, 475$, 476$, 477$, 478$, 479$, 480$, 481$, 482$, 483$, 484$, 485$, 486$, 487$, 488$, 489$, 490$, 491$, 492$, 493$, 494$, 495$, 496$, 497$, 498$, 499$, 500$, 501$, 502$, 503$, 504$, 505$, 506$, 507$, 508$, 509$, 510$, 511$, 512$, 513$, 514$, 515$, 516$, 517$, 518$, 519$, 520$, 521$, 522$, 523$, 524$, 525$, 526$, 527$, 528$, 529$, 530$, 531$, 532$, 533$, 534$, 535$, 536$, 537$, 538$, 539$, 540$, 541$, 542$, 543$, 544$, 545$, 546$, 547$, 548$, 549$, 550$, 551$, 552$, 553$, 554$, 555$, 556$, 557$, 558$, 559$, 560$, 561$, 562$, 563$, 564$, 565$, 566$, 567$, 568$, 569$, 570$, 571$, 572$, 573$, 574$, 575$, 576$, 577$, 578$, 579$, 580$, 581$, 582$, 583$, 584$, 585$, 586$, 587$, 588$, 589$, 590$, 591$, 592$, 593$, 594$, 595$, 596$, 597$, 598$, 599$, 600$, 601$, 602$, 603$, 604$, 605$, 606$, 607$, 608$, 609$, 610$, 611$, 612$, 613$, 614$, 615$, 616$, 617$, 618$, 619$, 620$, 621$, 622$, 623$, 624$, 625$, 626$, 627$, 628$, 629$, 630$, 631$, 632$, 633$, 634$, 635$, 636$, 637$, 638$, 639$, 640$, 641$, 642$, 643$, 644$, 645$, 646$, 647$, 648$, 649$, 650$, 651$, 652$, 653$, 654$, 655$, 656$, 657$, 658$, 659$, 660$, 661$, 662$, 663$, 664$, 665$, 666$, 667$, 668$, 669$, 670$, 671$, 672$, 673$, 674$, 675$, 676$, 677$, 678$, 679$, 680$, 681$, 682$, 683$, 684$, 685$, 686$, 687$, 688$, 689$, 690$, 691$, 692$, 693$, 694$, 695$, 696$, 697$, 698$, 699$, 700$, 701$, 702$, 703$, 704$, 705$, 706$, 707$, 708$, 709$, 710$, 711$, 712$, 713$, 714$, 715$, 716$, 717$, 718$, 719$, 720$, 721$, 722$, 723$, 724$, 725$, 726$, 727$, 728$, 729$, 730$, 731$, 732$, 733$, 734$, 735$, 736$, 737$, 738$, 739$, 740$, 741$, 742$, 743$, 744$, 745$, 746$, 747$, 748$, 749$, 750$, 751$, 752$, 753$, 754$, 755$, 756$, 757$, 758$, 759$, 760$, 761$, 762$, 763$, 764$, 765$, 766$, 767$, 768$, 769$, 770$, 771$, 772$, 773$, 774$, 775$, 776$, 777$, 778$, 779$, 780$, 781$, 782$, 783$, 784$, 785$, 786$, 787$, 788$, 789$, 790$, 791$, 792$, 793$, 794$, 795$, 796$, 797$, 798$, 799$, 800$, 801$, 802$, 803$, 804$, 805$, 806$, 807$, 808$, 809$, 810$, 811$, 812$, 813$, 814$, 815$, 816$, 817$, 818$, 819$, 820$, 8
```

```

structure (TABLE[CH, FIELD]) = * 2 + .FIELD * 2) < 0, 8 * 2);
structure (TABLE + .CH * NUM) = * 2 + .PTR * 2) < 0, 8 * 2);
structure (ARRY[CHNL, PTR]) = * 2 + .CHNL * IND
structure (ARRY + .CHNL * IND) = * 2 + .PTR * 2) < 0, 8 * 2);

```

```

external ACTIVE , DESCRIPTOR , HEADBLK ;
TABLE ACTIVE ; ! active table for all open files
ARRAY DESCRIPTOR ; ! descriptors of all open files
VECTOR HEADBLK ; ! stores header block in memory

```

```

00530 ENTERREQUEST - enters the disk request in disk manager's queue
00540 PROTECTK - does a protection check on file }
00550
00560 external GETBLK,PUTBLK; ! gets and puts free blocks on disk
00570 external GETDESC,PUTDESC; ! allocates and releases free descriptors
00580 external WAITSEM,SIGNALSEM;
00590 ! synchronizes the blocking and awakening on events
00600
00610
00620 MACRO OUTMSG(NUMB)=
00630 (RETURN (NUMB))$;
00640
00650
00660
00670
00680
00690

```

mode of access of tables ACTIVE & DESCRIPTOR := 1 ;

```

00010 MODULE SETUPINIT(NOLIST)=
00020 begin
00030 % this file contains two initializing routines
00040 % (1) SYSINIT: this is called when the system
00050 % is set up
00060 % (2) TMPINIT: this is called every time the
00070 % system is restarted
00080
00090 % author : AARTI KUMAR
00100 % date of creation : 24 JULY, 1982
00110
00120 MACRO HEADER = #000000$; ! block address of header
00130
00140 external INITROOT ! initializes the root directory
00150 INITSTORAGE ! creates a linked list of all free disk blks
00160 HEADINIT ! initializes a linked header block
00170 MAPINIT ! initializes the free descriptor map
00180 FILINIT ! initializes all global variables and arrays
00190 INITACTIVE ! initializes the active table
00200 INITSTAT ! initializes the table maintained by disk manager
00210 INITDISK ! initializes the disk controller
00220 DISKIO ! called to enter a disk-to request
00230
00240 external NJOB;
00250 external HEADBLK;
00260 external VECTOR HEADBLK;
00270
00280 GLOBAL ROUTINE SYSINIT(JOB)=
00290 begin
00300 % this routine is called when the system is set up %
00310
00320 NJOB = .JOB;
00330 INITDISK();
00340 INITSTAT();
00350 FILINIT();
00360 INITSTORAGE();
00370 INITROOT();
00380 MAPINIT();
00390 HEADINIT();
00400
00410 end;
00420
00430 GLOBAL ROUTINE TMPINIT(JOB)=
00440 begin
00450 % this routine is called each time the system restarts %
00460
00470 NJOB = .JOB;
00480 INITDISK();
00490 INITSTAT();
00500 INITACTIVE();
00510 FILINIT();
00520 DISKIO( 2, HEADBLK, HEADBLK);

```


00530
00540
00550
00560

end;

end ELUDOM

```

MODULE INITIALIZE(NOLIST)=
begin
% routine to initialize disk space %
% author : AARTI KUMAR %
% date of creation : 10 MAY , 1982 %

MACRO FILENAME = 0$,
JOBNO = 0$,
FILSTAT = 2$,
SIZE = 3$,
PARENT = 4$,
PREVDESC = 4$,
CURDESC = 5$,
NEXTDESC = 6$,
OWNER = 7$,
DATE = 8$,
STORAGE = 10$,
MAXM = 10$,
kind = 9,0,9$,
protection = 82$,
TRKFULL = 9$,
CYLFULL = #000000$,
HEADER = #100000$,
ROOT = 15,
latter = 15,
BLANK = $,
trk = 7,8$,
cylinder = 0,7$:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
field : <filename>
file : <jobno>
<file status word>
<size of file>
descriptor address of previous
decssor of current descriptor
address of next descriptor
file owner's ppn
file date of creation
storage map of file starts
maximum number of acyive files
{ 0/1 } -> directory/text file
protector of file
number of tracks in a cylinder
number of blocks in a block
address of header block
address of root directory
bit indicates position of descriptor in block
blank character)
track address field
cylinder address {logical} field
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

STRUCTURE TABLEch,field] = 2 + .field * 2)<0,8*2>;
STRUCTURE (.TABLE + .ch * 6 * 2 + .DFIELD * 2)<0,8*2>;
(.ARRAY + .DPNT * 64 * 2 + .DFIELD * 2)<0,8*2>;

external ACTIVE,DESCRIPTOR;
external GDATE;
map TABLE ACTIVE;
MAP ARRAY DESCRIPTOR;

external HEADBLK;
map VECTOR HEADDBLK;

external DISKIO, INTSTAT, INITDISK;
external NJOB;

external TRACEBUF,LINER;
map VECTOR TRACEBUF:LINER;

GLOBAL ROUTINE INITROOT=
```

```

00530 begin
00540   Local VALUE;
00550   % this routine initializes the root %
00560
00570   DISKIO(2, ROOT, TRACEBUF);
00580   TRACEBUF[FILENAME] = 'RO';
00590   TRACEBUF[FILENAME] + 1 J = 'OT';
00600   TRACEBUF[FILSTAT] <kind> = 1;
00610   TRACEBUF[FILSTAT] <protection> = #022;
00620   TRACEBUF[SIZE] = 0;
00630   TRACEBUF[PARENT] = ROOT;
00640   TRACEBUF[CURDESC] = ROOT;
00650   TRACEBUF[NEXTDESC] = -1;
00660   TRACEBUF[OWNER] = #000401;
00670   TRACEBUF[DATE] = GDATE();
00680   TRACEBUF[STORAGE] = '$';
00690   DISKIO(3, ROOT, TRACEBUF);
00700
00710 end;
00720
00730 GLOBAL ROUTINE INITSTORAGE =
00740 begin
00750   Local
00760     NEXT, CURRENT, BLK, I, J;
00770   % this creates a linked list of all free blocks on disk %
00780
00790   I = 80; BLK = 0;
00800   BLK <trk> = .1;
00810   BLK <cyl> = .J;
00820   CURRENT = BLK;
00830   NEXT = BLK + 1;
00840   INCR L FROM 0 TO 127 DO (LINEAR[L] = 0);
00850   do
00860     begin
00870       LINEAR[0] = NEXT;
00880       DISKIO(1, .CURRENT, LINEAR);
00890       CURRENT = NEXT;
00900       NEXT = NEXT + 1;
00910       if .NEXT <cylinder> eql CYLFULL then (NEXT <trk> = .NEXT <trk> + 1;
00920         NEXT <cylinder> = 0);
00930     end;
00940   until .NEXT <trk> eql (TRKFULL);
00950   NEXT = HEADER;
00960   LINEAR[0] = NEXT;
00970   DISKIO(1, .CURRENT, LINEAR);
00980   LINEAR[0] = .BLK;
00990   DISKIO(1, HEADER, LINEAR);
01000
01010 end;
01020 GLOBAL ROUTINE INITACTIVE =
01030 begin
01040   Local J;

```

```

01050      % this initializes the active table %
01060      J = 0;
01070      do begin
01080          INCR I FROM 1 TO 5 DO (ACTIVE[J,..I] = 0);
01090          ACTIVE[J,0] = -1; J = J+1;
01100          INCR I FROM 0 TO 63 DO
01110              (DESCRIPTOR[J,..I] = 0);
01120      end until (.J eq1 MAXM);
01130
01140      end;
01150
01160      GLOBAL ROUTINE HEADINIT=
01170      begin
01180          % this initializes the header block %
01190          DISKIO(2,HEADER,HEADBLK);
01200          HEADBLK[1] = ROOT;
01210          HEADBLK[2] = #000001;
01220          HEADBLK[3] = #000002;
01230          HEADBLK[4] = 179;
01240          HEADBLK[5] = 199;
01250          HEADBLK[6] = 4096;
01260          HEADBLK[7] = #000004;
01270          HEADBLK[8] = #000143;
01280          HEADBLK[9] = 96;
01290          DISKIO(3,HEADER,HEADBLK);
01300      end;
01310
01320      END ELUDOM
01330
01340
01350

```

!! address of root directory
 !! address of 1'st block of descriptor map
 !! address of 2'nd block of descriptor map
 !! starting track number of descriptors
 !! ending track number of descriptors
 !! number of blocks used to store descriptors
 !! starting block number of accounting file
 !! ending block number of accounting file
 !! number of blocks in accounting file

```

00530      end;
00540
00550      GLOBAL ROUTINE PUTBLK(JOB,VALUE) =
00560      begin
00570          Local ADDR2;
00580          % put a free block %
00590
00600          WAITSEM(1); PJOB = JOB;
00610          ADDR2 = HEADBLK[0];
00620          HEADBLK[0] = VALUE;
00630          BLKIO(3,HEADER,HEADRLK);
00640          BUFF1[0] = ADDR2;
00650          INCR I FROM 1 to 127 DO ( BUFF1[I,I] = ZERO );
00660          BLKIO(1,VALUE,BUFF1);
00670          SIGNALSEM(1);
00680      end;
00690
00700      end ELUDOM
00710
00720

```

```

00010 MODULE BITMAP(STACK,NOLIST)=
00020 begin
00030
00040 % this module contains routines for free descriptor management %
00050
00060 % AUTHOR : AARTI KUMAR %
00070 % DATE OF CREATION : 15 MAY , 82 %
00080
00090 BIND
00100 Header = #0000000,
00110 Addr1 = #0000001,
00120 Addr2 = #0000002;
00130
00140 MACRO
00150 ERRVAL = #077000$,
00160 oprn = 0.4$,
00170 FILBLK = 0.7$,
00180 TRKCYL = 7.8$,
00190 latter = 15.1$,
00200 lowlim = 179$,
00210 upperlim = 199$,
00220
00230 global BJOB;
00240
00250 MACRO OUTBIT(DIGIT)[I]=
00260 if not .BUFF2[I]<SLENGTH-1,1> then ( FLAG = 1; BUFF2[I]<SLENGTH-1,1> = 1)
00270 else ( COUNT = .COUNT+1; OUTBIT($REMAINING))$;
00280
00290 MACRO FILLBIT(VAL)[I]=
00300 if .BIT eq1 (SLENGTH-1) then ( (.TEMP)<SLENGTH-1,1> = 0 )
00310 else ( FILLBIT($REMAINING))$;
00320
00330 external ENTERREQUEST, DIVL,MODL;
00340 external WAITSEM,SIGNALSEM;
00350
00360 GLOBAL ROUTINE DESCIO(FUNC,DSKADDR,MEMLOCN)=
00370 begin
00380 % this routine enters request for diskio %
00390
00400 Local TASK,NEWLOCN;
00410
00420 TASK<oprn> = ,FUNC;
00430 if .FUNC leg 1 then
00440 begin
00450 NEWLOCN = .MEMLOCN;
00460 end;
00470 NEWLOCN = .MEMLOCN;
00480 ENTERREQUEST(.BJOB,.TASK,.DSKADDR,.NEWLOCN);
00490
00500 end;
00510
00520

```

```

00530 word own VECTOR BUFF2[128];
00540
00550 GLOBAL ROUTINE GETDESC(JOR)=
00560 begin
00570   Local TRKNO,COUNT,TEMP,ADDR,FLAG,BLKNO,WRD,BIT,I,VAL;
00580   % gets a free descriptor %
00590   WAITSEM(2); RJOB = .JOR;
00600   FLAG = 0; COUNT = 0; VAL = 0;
00610   ADDR = ADDR1;
00620   DESCIO(0,ADDR1,BUFF2); I = 0;
00630   while ((.COUNT leq 4095) and (not .FLAG))
00640   do begin
00650     TEMP = BUFF2[I];
00660     OUTBIT(16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1);
00670     I = I + 1;
00680     if (not (.FLAG and I eq 128)) then
00690       if (not (.ADDR = ADDR2; DESCIO(0,ADDR2,BUFF2); I = 0);
00700       end;
00710       if (not .FLAG) then (SIGNALSEM(2); RETURN -1);
00720       DESCIO(1,.ADDR,BUFF2);
00730       TRKNO = lowlim + DIVL(.COUNT,200);
00740       BLKNO = DIVL(.COUNT,200),2);
00750       VAL = 0;
00760       VAL<FILBLK> = .BLKNO;
00770       VAL<TRKCYL> = .TRKNO;
00780       if MODL(.COUNT,2) eq 1 then (VAL = .VAL + #1000000);
00790       INCR L FROM 0 TO 63 DO ( BUFF2[LJ]=0);
00800       DESCIO(3,.VAL,BUFF2);
00810       SIGNALSEM(2);
00820     end; .VAL
00830   end;
00840
00850 GLOBAL ROUTINE PUTDESC(JOR,VALUE) =
00860 begin
00870   Local TRKNO,COUNT,TEMP,ADDR,FLAG,BLKNO,WRD,BIT,I;
00880   % this routine puts a free descriptor %
00890
00900   WAITSEM(2); RJOB = .JOR;
00910   TRKNO = 0; BLKNO = 0;
00920   BLKNO = .VALUE<FILBLK>;
00930   TRKNO = .VALUE<TRKCYL>;
00940   COUNT = (.TRKNO - lowlim) * 200 + .BLKNO * 2;
00950   if .VALUE<latr> then COUNT = .COUNT + 1;
00960   if .COUNT leq 2045 then (ADDR = ADDR1) else (ADDR = ADDR2);
00970   DESCIO(0,.ADDR,BUFF2);
00980   WRD = DIVL(.COUNT,16);
00990   BIT = 15 - MODL(.COUNT, 16);
01000
01010
01020
01030
01040

```

```

01050
01060
01070
01080
01090
01100
01110
01120
01130
01140
01150
01160
01170
01180
01190
01200
01210

TEMP = BUFF2[1,WRD1];
FILLBIT(16,15;14,13;12,11,10,9,8,7,6,5,4,3,2,1);
DESCIO(1,ADDR,BUFF2);
SIGNALSEM(2);

end;

external DISKIO;
GLOBAL ROUTINE MAPINIT=
begin
    Incr I from 0 to 127 do
        begin BUFF2[I,I] = 0 end;
        DESCIO(1,ADDR1,BUFF2);
        DESCIO(1,ADDR2,BUFF2);
    end;
end ELUDOM

```



```

00010 MODULE MAKEDIRECTORY(NOLIST)=
00020 begin
00030
00040     % AUTHOP      : AARTI KUMAR
00050     % DATE OF CREATION : JUNE 5, 1982 %
00060
00070     % this primitive is used to create directory files
00080     % a message of successful completion is returned
00090     % unlike the creation of text files the file is not
00100     % activated here %
00110
00120
00130
00140
00150
00160
00170
00180
00190
00200
00210
00220
00230
00240
00250
00260
00270
00280
00290
00300
00310
00320
00330
00340
00350
00360
00370
00380
00390
00400
00410
00420
00430
00440
00450
00460
00470
00480
00490
00500
00510
00520

require FILER1,B11;
require FILER2,B11;

external GDATE,INITDESC,INTERVAL,FINDSPACE;

ROUTINE NEWDIR(ADDR,PATH,CHKLOCK,OWNR)=
% this routine initializes the descriptor for a new
% directory file %
begin
    DISKIO(2, ADDR, TRACEBUF);
    TRACEBUF[FILENAME] = (.PATH);
    TRACEBUF[FILENAME] + ii = (.PATH + 2);
    TRACEBUF[FILSTAT]<kind> = 1;
    TRACEBUF[FILSTAT]<protection> = .CHKLOCK;
    TRACEBUF[SIZE] = 0;
    TRACEBUF[PARENT] = .ADDRESS;
    TRACEBUF[CURDESC] = .ADDR;
    TRACEBUF[NEXTDESC] = -1;
    TRACEBUF[OWNER] = .OWNR;
    TRACEBUF[DATE] = GDATE();
    TRACEBUF[STORAGE] = EOLIST;
    DISKIO(3, .ADDR, TRACEBUF);
end;

GLOBAL ROUTINE CREDIR(JOB, PPN, FILNAM, CHKLOCK, OWRN)=
begin
    Local CURRENT, TEMP, PREVIOUS;

    % this is the file primitive "credir" used to creat
    % directory files : unlike "creator", the owner of
    % the directory must be given as a primitive %

    WAITSEM(0); ! check to wait on event 0
    NJOB= JOB; FDEROR = 0;
    if (.FILNAM) eq 1 then ADDRESS=ROOT else ADDRESS=(.FILNAM);
    PATH=.FILNAM + 2; TEMP = NJOB;
    ADDRESS =TRACE(.ADDRESS,.PPN); ! locate parent directory's address
end

```

```

00530 if (.ADDRESS .eq. ERRVAL) and (.ADDRESS lss MAXERR)
00540 then ( SIGNALSEM(0) ; RETURN .ADDRESS ) ; ! illegal path
00550 if not PROTCHK(,PPN,2,TRACEBUF(OWNER),TRACEBUF(FILESSTAT))
00560 then ( SIGNALSEM(0) ; OUTERR(8) ) ; ! file protected
00570 CURRENT = SEARCHDIPEC(.ADDRESS) ; ! search for file in directory
00580 if .CURRENT neq -1 then ( SIGNALSEM(0) ; OUTERR(3) ) ;
00590 ! file already exists
00600 DISKIO(2,ADDRESS,TRACEBUF) ;
00610 if not PROTCHK(,PPN,2,OWNER,CHKLOCK) then
00620 ( SIGNALSEM(0) ; OUTERR(8) ) ;
00630 ! no write access on directory-created space
00640 CURRENT = FINDSPACE(.ADDRESS) ; ! look for free space
00650 if .CURRENT neq -1 then ( LINEAR(.CURRENT * 3) eq. EOLIST
00660 then LINEAR(.CURRENT * 3 + 3) = EOLIST ; ! end of file
00670 STORE = ENTERVAL(.CURRENT * 3) ; ! create entry
00680 if (.STORE eq -1) then OUTERR(14) ;
00690 ( SIGNALSEM(0) ; OUTERR(14) ) ;
00700 DISKIO(1,TRACEBUF(STORE) ;
00710 DISKIO(2,ADDRESS,TRACEBUF) )
00720 else begin if .INDEX eq 54 then
00730 begin
00740 CURRENT = GETDESC(.NJOB) ;
00750 if (.CURRENT eq -1) then
00760 ( SIGNALSEM(0) ; OUTERR(14) ) ;
00770 PREVIOUS = TRACEBUF( CURDESC ) ;
00780 TRACEBUF( NEXTDESC ) = CURRENT ;
00790 DISKIO(3,PREVIOUS,TRACEBUF) ;
00800 INIDESC(.NJOB, .CURRENT, .PREVIOUS, TRACEBUF) ;
00810 INDEX = 0 ;
00820 end ;
00830 CURRENT = GETRLK(.NJOB) ; ! allocate a new block
00840 if (.CURRENT eq -1) then ( SIGNALSEM(0) ; OUTERR(15) ) ;
00850 DISKIO(0, .CURRENT, LINEAR) ;
00860 STORE = ENTERVAL(0) ; LINEAR(3) = EOLIST ;
00870 if (.STORE eq -1) then ( SIGNALSEM(0) ; OUTERR(14) ) ;
00880 ! create file entry in new block
00890 DISKIO(1, .CURRENT, LINEAR) ;
00900 TRACEBUF( STORE + .CURRENT ;
00910 DISKIO(1, .STORE + 3) then
00920 if (.INDEX lss 53) then
00930 (TRACEBUF(TRACEBUF(TRACEBUF( CURDESC ) then
00940 if .ADDRESS neq .begin
00950 DISKIO(3,TRACEBUF( CURDESC ) ;
00960 DISKIO(2,ADDRESS,TRACEBUF) ;
00970 end ;
00980 TRACEBUF( SIZE ) = TRACEBUF( SIZE ) + 1 ; ! modify size
00990 DISKIO(3,ADDRESS,TRACEBUF) ;
01000 end ;
01010 end ;
01020 CURRENT = OWNER ;
01030 NEWDIR(.STORE, .PATH, .CHKLOCK, .CURRENT) ; ! initializes descriptor
01040 end ;

```

```
01050      SIGNAL,SEM(0);  
01060      0  
01070      end;  
01080  
01090      end ELUDOM  
01100
```

```

00010 MODULE CREAT(NOLIST)=
00020 begin
00030
00040 % AUTHOR : AARTI KUMAR
00050 % DATE OF CREATION : MAY 25, 1982 %
00060
00070 % this is the file primitive used to create text files
00080 % it creates an entry in the appropriate file and
00090 % then activates the file like in open %
00100
00110 require FILER1.B11;
00120 require FILER2.B11;
00130
00140 external GDATE, FINDSPACE, INITDESC, INITCHNL, FINDCHNL, INTERVAL;
00150
00160 ROUTINE NEWDESC(ADDR, PATH, CHKLOCK, OWNR) =
00170 % this routine initializes a descriptor allocated to a
00180 % new text file %
00190
00200 begin
00210 DISKIO(2, ADDR, TRACEBUF);
00220 TRACEBUF[FILENAME] = (.PATH);
00230 TRACEBUF[FILENAME + 1] = (.PATH + 2);
00240 TRACEBUF[FILSTAT] <kind> = 0;
00250 TRACEBUF[FILSTAT] <protection> = .CHKLOCK;
00260 TRACEBUF[FILSIZE] = 0;
00270 TRACEBUF[PARENT] = .ADDRESS;
00280 TRACEBUF[CURDESC] = .ADDR;
00290 TRACEBUF[NEXTDESC] = -1;
00300 TRACEBUF[OWNER] = .OWNR;
00310 TRACEBUF[DATE] = GDATE();
00320 TRACEBUF[STORAGE] = '$';
00330 DISKIO(3, .ADDR, TRACEBUF);
00340
00350 end;
00360
00370
00380 GLOBAL ROUTINE CREATOR(JOB, PPN, FILNAM, CHKLOCK, BUFPTR) =
00390
00400 % this is the file primitive "CREATOR" which is called to
00410 % create text files %
00420
00430 begin
00440 Local TEMP, CURRENT, PREVIOUS;
00450
00460 WAITSEM(0); ! check to see if resource associated with event 0 is free
00470 NJOB = JOB; FERROR = 0;
00480 if (.FILNAM) eq -1 then ADDRESS=ROOT else ADDRESS=(.FILNAM);
00490 PATH=.FILNAM + 2;
00500 ADDRESS=TRACE(.ADDRESS, .PPN);
00510 ! this returns address of the directory in which file exists
00520 if (.ADDRESS qtr FERROR) and (.ADDRESS lss MAXERR)

```

```

00530 then ( SIGNALSEM(0); RETURN .ADDRESS );
00540 ! illegal path specification
00550 if not PROTCHK(.PPN,2,.TRACEBUF[OWNER],.TRACEBUF[FILSTAT])
00560 then ( SIGNALSEM(0); OUTERR(8) ); ! file protected against use
00570 CURRENT = SEARCHDIPEC(.ADDRESS);
00580 if (.CURRENT neq -1) then ( SIGNALSEM(0); OUTERR(3) );
00590 ! file already exists
00600 DISKIO(2,.ADDRESS,.TRACEBUF);
00610 if not PROTCHK(.PPN,2,.TRACEBUF[OWNER],.CHKLOCK) then
00620 ( SIGNALSEM(0); OUTERR(8) ); ! file cannot be opened for write
00630 CURRENT = FINDSPACE(.ADDRESS);
00640 if .CURRENT neq -1 then ( if .LINEAR[.CURRENT*3] = $;
00650 then .LINEAR[.CURRENT*3] = $; ! create file entry
00660 STORE = ENTVAL(.CURRENT*3); ! create file entry
00670 if (.STORE eq -1) then ( SIGNALSEM(0); OUTERR(14));
00680 DISKIO(1,.TRACEBUF[STORAGE] + .INDEX,LINEAR);
00690 DISKIO(2,.ADDRESS,TRACEBUF)
00700
00710 else begin
00720 if .INDEX eq 54 then
00730 begin
00740 ! allocate a new descriptor
00750 ! a .CURRENT = GETDESC(.NJOB);
00760 if (.CURRENT eq -1) then
00770 ( .SIGNALSEM(0); OUTERR(15) );
00780 PREVIOUS = .TRACEBUF[.CURRENT];
00790 TRACEBUF[.NEXTDESC] = .CURRENT;
00800 DISKIO(3,.PREVIOUS,TRACEBUF);
00810 INITDESC(.NJOB,.CURRENT,.PREVIOUS,TRACEBUF);
00820 INDEX = 0;
00830 end;
00840 CURRENT = GETBLK(.NJOB); ! allocate a new block
00850 if (.CURRENT eq -1) then ( .SIGNALSEM(0); OUTERR(14));
00860 DISKIO(0,.CURRENT,LINEAR);
00870 STORE = ENTVAL(0); LINEAR[3] = '$'; ! create file entry
00880 if (.STORE eq -1) then ( .SIGNALSEM(0); OUTERR(14) );
00890 if (.DISKIO[.CURRENT] + .INDEX) = .CURRENT;
00900 TRACEBUF[STORAGE] = .CURRENT;
00910 if (.INDEX lss 53) then
00920 (TRACEBUF[STORAGE] + .INDEX + 1) = '$' );
00930 if .ADDRESS neq begin
00940 DISKIO(3,.TRACEBUF[.CURRENT],TRACEBUF);
00950 DISKIO(2,.ADDRESS,TRACEBUF);
00960 end;
00970 TRACEBUF[SIZE] = .TRACEBUF[SIZE] + 1;
00980 ! modify size of directory file
00990 DISKIO(3,.ADDRESS,TRACEBUF);
01000
01010 end;
01020 CURRENT = .TRACEBUF[.OWNER];
01030 NEWDESC(.STORE,.PATH,.CHKLOCK,.CURRENT);
01040 CHANNEL = FINDCHNL();
    if (.CHANNEL eq -1) then ( .SIGNALSEM(0); OUTERR(9));

```

```

01050      INCR I to 63 do ( DESCRIPTOR(.CHANNEL, I) = TRACERUFF(.I) );
01060      INITCHANNEL(.CHANNEL, 1, .NJOB, .BUFPTR); ! initialize channel
01070      ACTIVE(.CHANNEL, FILSTAFF<extension> = .CHKLOCK<extension>);
01080      SIGNALSEM(0); ! signal release of resource associated with event 0
01090      .CHANNEL ! return the channel number
01100      end;
01110
01120
01130
01140      end ENJDM

```

```

00010 MODULE OPENFILE(NOLIST)=
00020 begin
00030
00040 % AUTHOR : AARTI KUMAR
00050 % DATE OF CREATION : JUNE 1, 1982 %
00060
00070 % this program is the file primitive OPEN
00080 % it takes as arguments PROCESS-NO, FILE-PATH-SPECIFICATION
00090 % USER'S PPN, FUNCTION such as READ/READ-WRITE & MEMORY
00100 % LOCATION in user's area from where each diskio request
00110 % is accomplished. %
00120
00130 % A CHANNEL NUMBER IS RETURNED TO THE USER
00140 % THIS MUST BE SPECIFIED FOR ANY FURTHER ACCESS ON
00150 % THE OPEN FILE %
00160
00170 require FILER1.B11;
00180 require FILER2.B11;
00190 external MATCH,INITCHNL,FINDCHNL;
00200
00210 GLOBAL ROUTINE OPEN(JOB, PPN,FILNAM,FUNCTION,BUFPTR)=
00220 begin
00230 local
00240 VALUE,TEMP;
00250
00260 begin
00270 WAITSEM(0); ! semaphore check for event 0
00280 NJOB = JOB; FERROR = 0;
00290 If (.FILNAM) eq 1 then
00300 ADDRESS=ROOT else ADDRESS=(.FILNAM);
00310 PATH=.FILNAM+2;ADDRESS,.PPN); ! trace gets descriptor
00320 ADDRESS=TRACE(.ADDRESS,.PPN); ! trace gets descriptor
00330 ! address of directory in which file exists
00340 If (.ADDRESS or ERROR) and (.ADDRESS lss MAXERR )
00350 then ( SIGNALSEM(0); RETURN .ADDRESS );
00360 ! illegal path specification
00370 if not PROTCHEK(.PPN,3,.TRACEBUF[OWNER],.TRACEBUF[FILSTAT] )
00380 then ( SIGNALSEM(0); OUTERR(8) );
00390 ! check for read access on directory
00400 ADDRESS = SEARCHDIREC(.ADDRESS); ! search for file in directory
00410 if .ADDRESS eq -1 then ( SIGNALSEM(0); OUTERR(2));
00420 ! error - file not found
00430 ! DISKIO(2, ADDRESS,TEMP=2 else TEMP = 3; .TRACEBUF[OWNER],.TRACEBUF[FILSTAT] )
00440 if .FUNCTION then TEMP=2 else TEMP = 3; .TRACEBUF[OWNER],.TRACEBUF[FILSTAT] )
00450 if not PROTCHEK(.PPN,.TEMP,.TEMP(0); OUTERR(8) );
00460 ! file protected against use
00470 then ( SIGNALSEM(0); OUTERR(5) ) )
00480 If .TRACEBUF[FILSTAT]<kind> then
00490 ! directory file cannot be opened for write
00500 else
00510 begin
00520

```

```

00530      If .FUNCTION then (if MATCH(.ADDRESS,2)
00540          then (SIGNALSEM(0); OUTERR(6)))
00550          ! file opened earlier
00560          else (if WATCH(.ADDRESS,1)
00570              then (SIGNALSEM(0); OUTERR(7)) )
00580              ! file opened for write earlier
00590              end;
00600          CHANNEL = FINDCHNL();
00610      If :CHANNEL = -1 then (SIGNALSEM(0); OUTERR (9) );
00620      ! no free channel available
00630      incr I to 63 do (DESCRIPTOR[ CHANNEL, I] = .TRACEBUF[ .I] );
00640      ! transfer file descriptor in specified channel
00650      ! INITCHNL(.CHANNEL, .FUNCTION, .JOB, .BUFFER);
00660      ACTIVEL(.CHANNEL, FILSTAT[extension] = .FUNCTION<extension>;
00670          SIGNALSEM(0); ! signal to semaphore for release of resource 0
00680          .CHANNEL
00690      end;
00700      end;
00710
00720      end ELUDOM
00730

```



```

00010 MODULE GETFIL(NOLIST)=
00020 begin
00030
00040 % file primitive to fetch a file block %
00050
00060 % author : AARTI KUMAR
00070 % date of creation : 1 JUNE , 1982 %
00080
00090 require FILER1.B11;
00100 EXTERNAL DIVL,MODL;
00110 EXTERNAL WRTDESC,FETCHDESC;
00120
00130
00140 GLOBAL ROUTINE GET(JOB,CH) =
00150 begin
00160 Local LOCN,FUNC,DESCNUM,BLOCK,OFFSET,POINTER,CHNL,CURNUMB;
00170
00180 % routine to get file blocks %
00190
00200 CHNL = .CH;
00210 if .JOB neq .ACTIVE1.CHNL,JOBNO1 then ABORT(.JOB,10);
00220 if .ACTIVE1.CHNL,CURBLK1 eq1 .ACTIVE1.CHNL,SIZE then ABORT(.JOB,12);
00230 DESCNUM = DIVL(.ACTIVE1.CHNL,CURBLK1,54);
00240 OFFSET = MODL(.ACTIVE1.CHNL,CURBLK1,54);
00250 CURNUMB = .ACTIVE1.CHNL,FILSTAT1<descno>;
00260 if .DESCNUM neq .CURNUMB then
00270 begin
00280 if .ACTIVE1.CHNL,FILSTAT1<curmod> then WRTDESC(.CHNL);
00290 if .ACTIVE1.CHNL,FILSTAT1<curmod> = 0;
00300 FETCHDESC(.DESCNUM,CURNUMB,.CHNL);
00310 end;
00320 BLOCK = .SCRIPTORI.CHNL,STORAGE + .OFFSET1;
00330 LOCN = .ACTIVE1.CHNL,BUFADDR1;
00340 FUNC<extension> = .ACTIVE1.CHNL,FILSTAT1<extension>;
00350 FUNC<oprin> = 0;
00360 ENTERREQUEST(.JOB,.FUNC,.BLOCK,.LOCN);
00370 .ACTIVE1.CHNL,CURBLK1 = .ACTIVE1.CHNL,CURBLK1 + 1; then OUTMSG(-1);
00380 if .ACTIVE1.CHNL,CURBLK1 eq1 .ACTIVE1.CHNL,SIZE then OUTMSG(-1);
00390 .ACTIVE1.CHNL,CURBLK1
00400 end;
00410
00420
00430
00440 end ELUDOM

```

```

00010 *JOBFILE PUTFIL(NOLIST)=
00020 begin
00030
00040 % file primitive to put blocks in a file %
00050
00060 % author : AARTI KUMAR
00070 % date of creation : 20 MAY , 1982 %
00080
00090 require FILE1.BLI;
00100 EXTERNAL DIVL,MODL;
00110 EXTERNAL WRTDESC,FLCHDESC;
00120 external INITDESC;
00130
00140
00150 GLOBAL ROUTINE PUT(JOB ,CH) =
00160 begin
00170 Local FUNC,MLOCN,FBLK,CHNL,OFFSET,DESCNUM,XADDR,YADDR,CURNUMB;
00180
00190 % routine to put blocks in a file %
00200
00210 CHNL = .CH;
00220 if .JOB neq .ACTIVE1.CHNL,JOBNO1 then ABORT(.JOB,10);
00230 if .not .ACTIVE1.CHNL,FILSTAT1<wrtbit> then ABORT(.JOB,11);
00240 DESCNUM = DIVL(.ACTIVE1.CHNL,CURBLK1,54);
00250 OFFSET = MODL(.ACTIVE1.CHNL,CURBLK1,54);
00260 CURNUMB = .ACTIVE1.CHNL,FILSTAT1<descno>;
00270 if .CURNUMB neq .DESCNUM then
00280 begin
00290 if .ACTIVE1.CHNL,FILSTAT1<curmod> then WRTDESC(.CHNL);
00300 if .ACTIVE1.CHNL,FILSTAT1<curmod> = 0;
00310 FETCHDESC(.DESCNUM,CURNUMB,.CHNL);
00320
00330 end; neq .DESCNUM then
00340 begin
00350 XADDR = GETDESC(.JOB);
00360 if .XADDR eq1 -1 then ABORT(.JOB,14);
00370 YADDR = .DESCRIPTOR1.CHNL,CURDESC1;
00380 DESCRIPTOR1.CHNL, NEXTDESCJ = .XADDR;
00390 WRTDESC(.CHNL);
00400 INITDESC(.JOB,.XADDR,YADDR,DESCRIPTOR1.CHNL,0);
00410 CURNUMB = .CURNUMB + 1;
00420 ACTIVE1.CHNL,FILSTAT1<descno> = .CURNUMB;
00430
00440 end;
00450 if .ACTIVE1.CHNL,CURBLK1 eq1 .ACTIVE1.CHNL,SIZE1
00460 then ( FBLK = GETBLK(.JOB);
00470 if .FBLK eq1 -1 then ABORT(.JOB,15) )
00480 else FBLK = .DESCRIPTOR1.CHNL,STORAGE + .OFFSET;
00490 MLOCN = .ACTIVE1.CHNL,BUFADDR1;
00500 FUNC<oprno> = 1;
00510 FUNC<extension> = .ACTIVE1.CHNL,FILSTAT1<extension>;
00520 ENTEREREQUEST(.JOB,.FUNC,.FBLK,.MLOCN);
00530 if .ACTIVE1.CHNL,CURBLK1 eq1 .ACTIVE1.CHNL,SIZE1 then

```

```

00530 begin DESCRIPTOR(.CHNL,STORAGE + .OFFSET) = .FRLK;
00540 if (.OFFSET) lss 53 then
00550 DESCRIPTOR(.CHNL,STORAGE + .OFFSET + 1) = FOLIST;
00560 ACTIVEL(.CHNL,FILSTAT1<CURMOD>=1;
00570 ACTIVEL(.CHNL,FILSTAT1<FIRSTMOD>=1;
00580 ACTIVEL(.CHNL,SIZE) = .ACTIVEL(.CHNL,SIZE) + 1;
00590 end;
00600 ACTIVEL(.CHNL,CURBLK) = .ACTIVEL(.CHNL,CURBLK) + 1;
00610 if .ACTIVEL(.CHNL,CURBLK) eq1 .ACTIVEL(.CHNL,SIZE) then UUIMSG(-1);
00620 .ACTIVEL(.CHNL,CURBLK)
00630 end;
00640
00650
00660
00670 end ELUDOM

```

00010 MODULE PUTMRKEUF(NOLIST)=
00020 begin

00030 % puts an end of file mark in the storage map %
00040
00050 % author : AARTI KUMAR
00060 % date of creation : 5 JUNE, 1982 %
00070
00080

00090 require FILE1.B11;
00100 EXTERNAL DIVL,MODL;
00110 EXTERNAL WRTDESC,FETCHDESC;

00120 GLOBAL ROUTINE PUTEUF(JOB,CHNL) =
00130 begin
00140 local DESCNUM,ADDR,COUNT,OFFSET,PTR,CURNUMB;

00150 % routine called to put an end of file mark %
00160
00170
00180
00190

00200 if .ACTIVE(.CHNL,JOBNO) neq .JOB then ABORT(.JOB,10);
00210 if not .ACTIVE(.CHNL,FILSTAT) < wrtbit> then ABORT(.JOB,11);
00220 if .ACTIVE(.CHNL,CURBLK) eql .ACTIVE(.CHNL,SIZE) then RETURN 0;
00230 DESCNUM = DIVL(.ACTIVE(.CHNL,CURBLK),54);
00240 OFFSET = MODL(.ACTIVE(.CHNL,CURBLK),54);
00250 CURNUMB = .ACTIVE(.CHNL,FILSTAT) < descno>;
00260 if .DESCNUM neq .CURNUMB then
00270 begin

00280 if .ACTIVE(.CHNL,FILSTAT) < curmod> then WRTDESC(.CHNL);
00290 .ACTIVE(.CHNL,FILSTAT) < curmod> = 0;
00300 FETCHDESC(.DESCNUM,CURNUMB,.CHNL);
00310

00320 end;
00330 COUNT = 0;

00340 Incr I from 0 to (53 - .OFFSET) do
00350 begin

00360 if .DESCRIPTOR(.CHNL,STORAGE + .OFFSET + .I) eql EOLIST
00370 then EXITLOOP;
00380 PUTBLK(.JOB,.DESCRIPTOR(.CHNL,STORAGE + .OFFSET + .I));
00390 COUNT = .COUNT + 1;

00400 end;
00410 if (.OFFSET eql 0) and (.CURNUMB gtr 0)

00420 then begin
00430 ADDR = .DESCRIPTOR(.CHNL,PREVDESC);
00440 PUTDESC(.JOB,.DESCRIPTOR(.CHNL,CURDESC));
00450 ENTERREQUEST(.JOB,2,ADDR,DESCRIPTOR(.CHNL,0));
00460 .DESCRIPTOR(.CHNL,NEXTDESC) = -1;
00470 .ACTIVE(.CHNL,FILSTAT) = .CURNUMB - 1;
00480 if .ACTIVE(.CHNL,FILSTAT) < descno> neq 0
00490 then WRTDESC(.CHNL);

00500 end
00510 else begin

00520 .DESCRIPTOR(.CHNL,STORAGE + .OFFSET) = EOLIST;
00530 if .COUNT eql (54 - .OFFSET) then

```

00530 begin = .DESCRIPTOR(.CHNL,NEXTDESC);
00540 ADDR = .ADDR neq -1 then
00550 if .ADDR neq -1 then
00560 begin
00570   .DESCRIPTOR(.CHNL,NEXTDESC) = -1;
00580   WRITDESC(.CHNL);
00590   PTR = .DESCRIPTOR(.CHNL,CURDESC);
00600   do
00610     begin
00620       ENTERREQUEST(.JOB, 2, .ADDR,DESCRIPTOR(.CHNL,0));
00630       incr I first 0 to 53 do
00640         (if .DESCRIPTOR(.CHNL, STORAGE + .I )
00650           eq1 ENLIST then EXITLOOP;
00660           PUTBLK(.JOB,.DESCRIPTOR(.CHNL, STORAGE + .I ));
00670           COUNT = .COUNT + 1;
00680           PUTDESC(.JOB,.ADDR);
00690           ADDR = .DESCRIPTOR(.CHNL, NEXTDESC);
00700         end
00710       until (.ADDR eq1 -1);
00720       ENTERREQUEST(.JOB, 2,.PTR,DESCRIPTOR(.CHNL,0));
00730     end;
00740   end;
00750   .ACTIVE[.CHNL,SIZE] = .ACTIVE[.CHNL,SIZE] - .COUNT;
00760   .ACTIVE[.CHNL,FILSTAT]<firstmod> = 1;
00770   .ACTIVE[.CHNL,FILSTAT]<curmod> = 0;
00780   .COUNT
00790 end;
00800
00810
00820
00830 end ELUDOM

```

```

00010 MODULE SEEKBLK(NOLIST)=
00020 begin
00030   % file primitive to set the current block pointer %
00040   %
00050   % author : AARTI KUMAR
00060   % date of creation : 22 MAY , 1982 %
00070   require FILER1.R11;
00080
00090   GLOBAL ROUTINE SEEK(JOB,CHNL,FBLK) =
00100   begin
00110     % current block pointer is set at value specified %
00120     %
00130     if .JOB neq .ACTIVE[,CHNL,JOBNO] then ABORT(.JOB,10);
00140     if .FBLK geq .ACTIVE[,CHNL,SIZE] then
00150       begin
00160         ACTIVE[,CHNL,CURBLK] = .ACTIVE[,CHNL,SIZE] ; file
00170         OUTMSG(-1); !message to indicate end of file
00180       end
00190     else ACTIVE[,CHNL,CURBLK] = .FBLK;
00200   end; .FBLK
00210
00220
00230
00240
00250 end ELUDOM

```

```

00010  MODULE BLKPTRCHK(WULIST)=
00020  begin
00030      % file primitive to check the current block pointer %
00040      %
00050      % author : AARTI KUMAR
00060      % date of creation : 22 MAY , 1982 %
00070
00080      require FILER1.b11;
00090
00100      GLOBAL ROUTINE BLKCHK(JOB,CHNL)=
00110      begin % returns the current block pointer value %
00120          if .JOB neq .ACTIVE[.CHNL,JOBNO] then ABORT(.JOB,10);
00130          if .ACTIVE[.CHNL,CURBLK] eq1 .ACTIVE[.CHNL,SIZE]
00140              then OUTMSG(-1); ! end of file indicated
00150          .ACTIVE[.CHNL,CURBLK]
00160      end;
00170
00180      end ELUDOM
00190
00200
00210

```

```

00010 MODULE CLOSEFILE(STACK,NOLIST)=
00020 begin
00030   % file primitive to deactivate files %
00040   %
00050   % author : AARTI KUMAR
00060   % date of creation : 25 MAY , 1982 %
00070
00080 require FILER1.R11;
00090
00100 external WRTDESC;
00110
00120 GLOBAL ROUTINE CLOSE(JOB,CH) =
00130 begin
00140   Local POINTER,CHNL,ADDR,CURNT;
00150
00160   % routine to close files %
00170
00180   CHNL = CH;
00190   if .ACTIVEI.CHNL,JORNOI neq .JOB then ABORT(.JOB,10);
00200   if .ACTIVEI.CHNL,FILSTATJ<delbits> eq 1 then
00210     begin
00220       if .ACTIVEI.CHNL,FILSTATJ<descno> neq 0 then
00230         (ENTERREQUEST(.JOB,2,.ACTIVEI.CHNL,PARENTJ,DESCRIPTORI.CHNL,0J));
00240       do begin
00250         Incr I from 0 to 53 do
00260           begin
00270             if .DESCRIPTORI.CHNL,STORAGE + .I eq EOLIST
00280               then EXITLOOP else
00290                 (PUTBLK(.JOB,.DESCRIPTORI.CHNL,STORAGE + .I) );
00300             end;
00310             CURNT = .DESCRIPTORI.CHNL,NEXTDESCJ;
00320             WRTDESC(.JOB,.DESCRIPTORI.CHNL,CURDESCJ);
00330             if .CURNT neq -1 then
00340               ( ENTERREQUEST(.JOB,2,.CURNT,DESCRIPTORI.CHNL,0J) );
00350             end
00360             until .CURNT eq -1;
00370           end
00380         else
00390           if .ACTIVEI.CHNL,FILSTATJ<delbits> gtr 1 then
00400             begin
00410               Incr I from 0 to (MAXM-1) do
00420                 (if .ACTIVEI.CHNL,PARENTJ eq 1 .ACTIVEI.I,PARENTJ then
00430                   (.ACTIVEI.I,FILSTATJ<delbits> =
00440                     .ACTIVEI.CHNL,PARENTJ<delbits> - 1 ) );
00450               end
00460             else begin
00470               .ACTIVEI.CHNL,FILSTATJ<curmod> and
00480               ( .ACTIVEI.CHNL,FILSTATJ<descno> neq 0 ) ) then
00490                 begin
00500                   WRTDESC(.CHNL);
00510                   .ACTIVEI.CHNL,FILSTATJ<curmod> = 0;
00520                   if .ACTIVEI.CHNL,FILSTATJ<firstmod>

```



```

00530      then ( ADDR = .ACTIVE1.CHNL,PARENT1;
00540            ENTERQUEST(.JOB,2,.ADDR,DESCRPTOR1.CHNL,01);
00550            ACTIVE1.CHNL,FILSTAT)<descno> = 0 );
00560      end;
00570      if .ACTIVE1.CHNL,FILSTAT)<firstmod> then
00580      begin
00590          DESCRPTOR1.CHNL,SIZE) = .ACTIVE1.CHNL,SIZE) ;
00600          WPIDESC(.CHNL);
00610      end;
00620      end;
00630      ACTIVE1.CHNL,JURNO) = -1;
00640      incr I from 1 to 5 do (ACTIVE1.CHNL,.IJ = 0 );
00650      0
00660      end;
00670
00680      end ELUDOM
00690

```

```

00010 MODULE CUPDIRPR(MOLIST)=
00020 begin
00030   % file primitive to set the current directory pointer %
00040   %
00050   % author : AARTI KUMAR
00060   % date of creation : 10 JUNE, 1982 %
00070
00080   require FILER1.B11;
00090   require FILER2.B11;
00100
00110 GLOBAL ROUTINE CDPCHK( JOB,PPN,FILNAM) =
00120 begin
00130   % this routine returns descriptor address of directory %
00140
00150   WAITSEM(0);
00160   NJOB = .JOB; FDEPROR = 0;
00170   PATH = .FILNAM; ADDRESS = .ROOT;
00180   ADDRESS = TRACE(.ADDRESS,.PPN);
00190   if (.ADDRESS <= 0) then (.ADDRESS = MAXERR) then
00200     ( SIGNALSEM(0); RETURN ADDRESS );
00210   if not PKOTCHK(.PPN,3,.TRACEBUF[OWNER],.TRACEBUF[FILSTAT] )
00220     then ( SIGNALSEM(0); OUTERR(8) );
00230   ADDRESS = SEARCHDIRREC(.ADDRESS);
00240   if ADDRESS = -1 then ( SIGNALSEM(0); OUTERR(2) );
00250   DISKIO(2, ADDRESS, TRACEBUF);
00260   if not (.TRACEBUF[FILSTAT]<kind>) then ( SIGNALSEM(0); OUTERR(4));
00270   SIGNALSEM(0);
00280
00290   end;
00300
00310   end ELUDOM
00320

```

```

00010 MODULE PENAMFILE(NOLIST)=
00020 begin
00030   % file primitive to rename files and change protection %
00040   % author : AARTI KUMAR
00050   % date of creation : 15 JUNE, 1982 %
00060
00070
00080 require FILER1.R11;
00090 require FILER2.R11;
00100
00110 GLOBAL ROUTINE RENAME(JOB,PPN,FILNAM,NEWLOCK)=
00120 begin
00130   Local FLAG,J,CURRENT,PTEMP,STEMP;
00140
00150   % this routine is called to rename or change protection
00160   % for a file %
00170
00180   WAITSEM(0);
00190   NJOB= JOB; FERROR= 0;
00200   If (.FILNAM) eq 1 then ADDRESS = ROOT else ADDRESS = (.FILNAM);
00210   PATH= (.FILNAM) + 2;
00220   ADDRESS= TRACE(.ADDRESS,.PPN) ; (.ADDRESS lss MAXERR)
00230   If (.ADDRESS lss MAXERR) then RETURN ADDRESS;
00240   If then ( SIGNALSEM(0) );
00250   If not ( PROCHK(.PPN,2,TRACEBUF[OWNER],TRACEBUF[FILSTAT]) )
00260   then ( SIGNALSEM(0); OUTERR(8) );
00270   If ( (.PATH + 6) neq -1 ) then
00280     begin
00290       PTEMP = .PATH + 4;
00300       PATH = .PATH + 4;
00310       (.PATH) = .ADDRESS;
00320       STEMP = SEARCHDIRREC(.ADDRESS);
00330       If .STEMP neq -1 then ( SIGNALSEM(0); OUTERR(1) );
00340       PATH = .PTEMP;
00350       (.PATH + 4) = EOLIST;
00360     end;
00370
00380     CURRENT = SEARCHDIRREC(.ADDRESS);
00390     If .CURRENT eq 1 then ( SIGNALSEM(0); OUTERR(2) );
00400     DISKIO(2,CURRENT,TRACEBUF);
00410     If not ( PROCHK(.PPN,1,TRACEBUF[OWNER],TRACEBUF[FILSTAT]) )
00420     then ( SIGNALSEM(0); OUTERR(8) );
00430     If ( (.PATH + 6) neq -1 ) then
00440       begin
00450         FLAG = 0; J = 0;
00460         do begin
00470           if ( ( LINEARI,J*31 eq 1 (.PATH)) and
00480             ( LINEARI,J*31+1 eq 1 (.PATH+2)) )
00490             then FLAG = 1 else J = .J+1;
00500         end
00510         until ( .J eq 1 MAXM) or ( .FLAG);
00520         PATH = .PATH + 6;

```

```

00530 LINEAR[J * 3] = (.PATH);
00540 LINEAR[J * 3 + 1] = (.PATH + 2);
00550 DISKIO(1,STORE,LINEAR);
00560 TRACEBUF[FILENAME] = (.PATH);
00570 TRACEBUF[FILENAME + 1] = (.PATH + 2);
00580 end;
00590 If .NEWLOCK neq -1 then
00600 begin
00610   if .PPN neq TRACEBUF[OWNER] then
00620     ( if not PKIOCHK(.PPN,0,TRACEBUF[OWNER],TRACEBUF[FILSTAT])
00630       then ( SIGNALSEM(0); OUTERR(8) );
00640       TRACEBUF[FILSTAT]<protection> = .NEWLOCK
00650     end;
00660     DISKIO(3,.CURRENT,TRACEBUF);
00670     Incr I from 0 to (MAXM -1) do
00680       begin
00690         if .ACTIVE[I,PARENT] eq1 .CURRENT then
00700           begin
00710             if .ACTIVE[I,FILSTAT]<descno> eq1 0
00720               then ( DESCRIPTOR[I,FILENAME] = .TRACEBUF[FILENAME];
00730                 DESCRIPTOR[I,FILENAME + 1] = .TRACEBUF[FILSTAT];
00740                 end;
00750             end;
00760             SIGNALSEM(0);
00770             end;
00780             end;
00790             end;
00800             end;
00810             end;
00820             end;
00830             end;

```

```

00010 MODULE DELETEFILE(EOLIST)=
00020 begin
00030   % file primitive delete files %
00040   author : AARTI KUMAR
00050   % date of creation : 22 JUNE , 1982 %
00060
00070 require FILE1.R11;
00080 require FILE2.R11;
00090
00100 external MATCH;
00110
00120 ROUTINE FREEENTRY(POSITION)=
00130 begin
00140   Local DELSIZE,TEMP,FLAG,I;
00150   % this routine frees the file entry from the directory %
00160
00170   If (.LINEAR[.POSITION * 3 + 3] eql EOLIST) or (.POSITION eql 41)
00180   then begin
00190     If (.POSITION eql 41)
00200     then (If .TRACEBUF[STORAGE + .INDEX + 1] neq EOLIST
00210           then begin
00220             LINEAR[.POSITION * 3] = -1;
00230             LINEAR[.POSITION * 3 + 1] = 0;
00240             LINEAR[.POSITION * 3 + 2] = 0;
00250             DISKIO(1,.STORE,LINEAR);
00260             RETURN
00270           end)
00280           ;
00290           DELSIZE = FLAG = 0; I = 1;
00300           While not .FLAG do
00310             begin
00320               While (.POSITION - .I) geq 0 do
00330                 begin
00340                   If .LINEAR[(.POSITION - I) * 3] neq -1 then
00350                     If (.LINEAR[(.POSITION - I) * 3 + 3] = EOLIST;
00360                         DISKIO(1,.STORE,LINEAR);
00370                         FLAG = 1; EXITLOOP)
00380                     else I = .I + 1;
00390                 end;
00400             end;
00410             If (.POSITION - .I) lss 0 then
00420               begin
00430                 PUTBLK(.NJOB,.STORE);
00440                 DELSIZE = DELSIZE + .INDEX;
00450                 TRACEBUF[STORAGE + .INDEX] = EOLIST;
00460                 If .INDEX neq 0 then
00470                   (INDEX = INDEX - 1;
00480                     STORE = .TRACEBUF[STORAGE + .INDEX];
00490                     DISKIO(0,.STORE,LINEAR);
00500                     POSITION = 41;
00510                     I = 0; FLAG = 0)
00520

```

```

00530 else begin
00540   if ADDRESS neq TRACEBUF[CURDESC1] then
00550     (TEMP = TRACEBUF[PREVDESC1];
00560     GUIDESC((NJOB, TRACEBUF[CURDESC1]);
00570     DISKIO(2, TEMP, TRACERUF);
00580     INDEX = 53;
00590     POSITION = 41; I = 0; FLAG = 0)
00600   else (FLAG = 1);
00610   end;
00620   ena;
00630
00640   if TRACEBUF[CURDESC] neq ADDRESS then
00650     begin
00660       DISKIO(3, TRACEBUF[CURDESC], TRACERUF);
00670       DISKIO(2, ADDRESS, TRACERUF);
00680     end;
00690     TRACEBUF[SIZE] = TRACEBUF[SIZE] - DELSIZE;
00700     DISKIO(3, ADDRESS, TRACEBUF);
00710   end
00720   else begin
00730     LINEAR[POSITION * 3] = -1;
00740     LINEAR[POSITION * 3 + 1] = 0;
00750     LINEAR[POSITION * 3 + 2] = 0;
00760     DISKIO(1, STORE, LINEAR);
00770   end;
00780
00790
00800
00810
00820
00830
00840
00850
00860
00870
00880
00890
00900
00910
00920
00930
00940
00950
00960
00970
00980
00990
01000
01010
01020
01030
01040

```

GLOBAL ROUTINE DELETE(JOB,PPN,FILNAM)=
begin
local POS,COUNT,I,J,CURRENT;
% this routine is called to delete files %
WAITSEM(0);
NJOB = JOB; FERROR = 0;
if (.FILNAM) eq 1 -1 then ADDRESS = ROOT else ADDRESS = (.FILNAM);
PATH = FILNAM + 2;
ADDRESS = TRACE(ADDRESS,PPN);
if (.ADDRESS qtr ERRVAL) and (.ADDRESS 1ss MAXERR) then
(SIGNALSEM(0)); RETURN ADDRESS;
if not PROCHK(PPN,2,TRACEBUF[OWNER],TRACEBUF[FILSTAT])
then (SIGNALSEM(0); OUTERR(8));
CURRENT = SEARCHDIREC(ADDRESS);
if DISKIO(2,CURRENT,TRACEBUF);
if not PROCHK(PPN,1,TRACEBUF[OWNER],TRACEBUF[FILSTAT])
then (SIGNALSEM(0); OUTERR(8));
if TRACEBUF[FILSTAT]<kind> then (SIGNALSEM(0); OUTERR(16));
if TRACEBUF[SIZE] qtr 0 then (SIGNALSEM(0); OUTERR(16));
if MATCH(CURRENT,2) then

```

01050 begin I = 0; J = 0;
01060 while .LSS MAXM do
01070 (if .ACTIVE(I,PARENT) eq1 .CURRENT
01080 then (J = .J + 1); I = .I + 1);
01090 INCR K from 0 to (MAXM - 1) do
01100 (if .ACTIVE(K,PARENT) eq1 .CURRENT then
01110 .ACTIVE(K,FILESTAT)<delbits> = .J );
01120 COUNT = .TRACEBUF(SIZE);
01130 end
01140
01150 else begin
01160 COUNT = 0;
01170 do
01180 begin
01190 DISKIO(2,.CURRENT,TRACEBUF);
01200 INCR I from 0 to 53 do
01210 begin
01220 if .TRACEBUF(STORAGE+.I) eq1 FOLIST
01230 then EXITLOOP else
01240 (PUTBLK(.NJOB,.TRACEBUF(STORAGE+.I));
01250 COUNT = .COUNT + 1);
01260 end;
01270 CURRENT = .TRACEBUF(NEXTDESC);
01280 PUTDESC(.NJOB,.TRACEBUF(CURDESC));
01290 end
01300 until .CURRENT eq1 -1;
01310 end;
01320 DISKIO(2,.ADDRESS,TRACEBUF);
01330 CURRENT = SEARCHDIRCC(.ADDRESS);
01340 POS = ( INCR I from 0 to 41 do ( if ( (.LINEAR(I*3) eq1 ( (.PATH) )
01350 and ( (.LINEAR(I*3+1) eq1 ( (.PATH + 2) ) ) )
01360 then EXITLOOP .I );
01370 FREENTRY(.POS);
01380 SIGNALSEM(0);
01390 end;
01400 COUNT
01410
01420 END ELUDOM

```

```

00010 MODULE ROUTES(MOLIST) =
00020 begin
00030 % common routines for the filing system %
00040 %
00050 % author : AARTI KUMAR
00060 % date of creation : 10 MAY , 1982 %
00070
00080 require FILE1.R11;
00090 external TRACFBUF,LINEAP;
00100 %
00110 % VECTOR TRACFBUF , VECTOR LINEAP;
00120
00130 external STORE, INDEX, PATH, ADDRESS, CHANNEL, NJDR, FERRORR;
00140 external DISKIO;
00150
00160 MACRO JUTERR(NUMB) =
00170 ( FERRORR = .NUMB + EPRVAL ; RETURN -1 )$;
00180
00190 GLOBAL ROUTINE FETCHDESC(NUMBER,CURNUMB,CHNL)=
00200 begin
00210 Local BOOL, FBLK;
00220
00230 % this routine fetches the right descriptor in memory %
00240
00250 BOOL = 0;
00260 do
00270 begin
00280 if (.CURNUMB ) GTR .NUMBER then ( (.CURNUMB) = (.CURNUMB) - 1;
00290 FBLK = DESCRIPTOR(CHNL,PREVDESC) )
00300 else ( (.CURNUMB) = (.CURNUMB) + 1;
00310 FBLK = DESCRIPTOR(CHNL,NEXTDESC) );
00320
00330 if .FBLK eq 1 then ( (.CURNUMB) = (.CURNUMB) - 1;
00340 BOOL = 1;
00350 FBLK = DESCRIPTOR(CHNL,CURDESC) )
00360 else ( ENTERREQUEST(.ACTIVE,CHNL,JOBN01,2,.FBLK,DESCRIPTOR(CHNL,01)) );
00370
00380 end
00390 until ( (.CURNUMB) eq 1 .NUMBER ) OR .BOOL;
00400 Active(CHNL,FILSTAT)<descno> = (.CURNUMB );
00410
00420 end;
00430
00440 GLOBAL ROUTINE WRTEDESC(CHNL) =
00450 begin
00460 Local ADDR;
00470 % this routine writes back the descriptor from a specified
00480 % channel %
00490
00500 ADDR = .DESCRIPTOR(CHNL,CURDESC);
00510 ENTERREQUEST(.ACTIVE,CHNL,JOBN01,3,.ADDR,DESCRIPTOR(CHNL,01));
00520

```



```

00530 end;
00540
00550 GLOBAL ROUTINE INTERVAL(POSITION) =
00560 begin
00570   % this enters a directory entry %
00580
00590   LINEAR(.POSITION) = (.PATH);
00600   LINEAR(.POSITION + 1) = (.PATH + 2);
00610   LINEAR(.POSITION + 2) = GETDESC(.JOB);
00620
00630   .LINEAR(.POSITION + 2)
00640
00650 end;
00660
00670 GLOBAL ROUTINE FINDSPACE(ADDRESS) =
00680 begin
00690
00700   % this finds a free space in the directory %
00710
00720   STORE=INDEX=0;
00730   STORE = .TRACEBUF[ STORAGE];
00740   while (.STORE neq ENLIST) do
00750     begin
00760       DISKIO(0, .STORE, LINEAR);
00770       (incr i to 41 do
00780         if (.LINEAR[i] * 3] eq 1) or
00790           (.LINEAR[i] * 31 eq 1 ENLIST)
00800           then RETURN .i);
00810       INDEX = INDEX + 1;
00820       if (.INDEX eq 54) then (ADDRESS = .TRACEBUF[ NEXTDESC];
00830         if ADDRESS eq 1 -1 then RETURN -1;
00840         DISKIO(2, .ADDRESS, TRACEBUF);
00850         INDEX = 0);
00860       STORE = .TRACEBUF[ STORAGE + .INDEX];
00870
00880     end;
00890
00900   -1
00910 end;
00920
00930 GLOBAL ROUTINE INITDESC(JOB, CURRENT, PREVIOUS, BUFFER) =
00940 begin
00950
00960   % initializes second and later descriptors %
00970
00980   ENTERREQUEST(.JOB, 2, .CURRENT, BUFFER);
00990   (.BUFFER)[ PREVDESC] = .PREVIOUS;
01000   (.BUFFER)[ CURDESC] = .CURRENT;
01010   (.BUFFER)[ NEXTDESC] = -1;
01020   ENTERREQUEST(.JOB, 3, .CURRENT, .BUFFER);
01030
01040 end;

```

```

01050 GLOBAL ROUTINE MATCH(DSKADD,FUNC) =
01060 begin
01070 local ROOT,I;
01080
01090 * this routine looks for a match of file in active table %
01100
01110 ROOT = 0; I = 0;
01120 while (not .ROOT) and (.I lss MAXM) do
01130 begin
01140 if .ACTIVE(I,PARENT) eq1 .DSKADD then
01150 select .FUNC of
01160
01170 1: if not .ACTIVE(I,FILSTAT)<wrthbit> then ROOT = 1;
01180 1: if .ACTIVE(I,FILSTAT)<wrthbit> then ROOT = 1;
01190
01200 otherwise: ROOT = 1;
01210 if not;
01220 I = .I + 1;
01230
01240
01250
01260
01270
01280
01290
01300
01310 end;
01320
01330 end;
01340
01350
01360
01370
01380
01390
01400
01410
01420
01430
01440
01450
01460
01470
01480
01490
01500
01510
01520
01530
01540
01550
01560

```

```

GLOBAL ROUTINE SEARCHDIPEC(FILADDR) =
begin
% this searches for a file in a directory %

STORE = INDEX=0;
STORE = .TRACEBUF[ STORAGE 1;
while
begin
DISKIO(0, .STORE, LINEAR);
( incr I (if .LINEAR(I*3) eq1 (.PATH)
and .LINEAR(I*3 + 1) eq1 (.PATH + 2)
then ( RETURN .LINEAR(I*3 + 2) );
if .LINEAR(I*3) eq1 EOLIST then (RETURN -1)));
INDEX = .INDEX + 1;
if (.INDEX eq1 54) then (FILADDR = .TRACEBUF[ NEXTDESC];
if .FILADDR eq1 -1 then RETURN -1;
DISKIO(2, .FILADDR, TRACEBUF);
INDEX = 0);
STORE = .TRACEBUF[ STORAGE + .INDEX];
end;

-1

end;

GLOBAL ROUTINE TRACE(ADDR,PPN) =
begin

```

```

01570 % this routine traces the directory containing
01580 % the file from the point specified &
01590
01600 while (.PATH + 4) neq EOLIST do
01610 begin
01620     DISKIO(2, ADDR, TRACERUF);
01630     if not PRDCHK(.PPR,3,.TRACERUF,UNIT1,.TRACERUF[FILSTAT]<protection>)
01640     then OUTERR(9);
01650     if not .TRACERUF[FILSTAT]<kind> then OUTERR(1);
01660     ADDR = SEARCHDIRFC(.ADDR);
01670     if .ADDR eq 1 then OUTERR(1);
01680     PATH = .PATH + 4;
01690 end;
01700
01710 DISKIO(2,.ADDR,TRACERUF);
01720 if not .TRACERUF[FILSTAT]<kind> then OUTERR(1);
01730 .ADDR
01740
01750 end;
01760
01770 GLOBAL ROUTINE FINDCHNL =
01780 begin
01790     Local I;
01800     % this routine looks for a free channel &
01810
01820     I = 0;
01830     while .I lss MAXM do ( if .ACTIVE[I,0] eq 1 -1
01840     then RETURN .I else I = .I + 1 );
01850     -1
01860
01870 end;
01880
01890 GLOBAL ROUTINE INITCHNL(CHANNEL,FUNCTION,JOB,BUFPTR) =
01900 begin
01910     % this initializes a channel &
01920
01930     ACTIVE[.CHANNEL,CURBLK] = .JOB;
01940     ACTIVE[.CHANNEL,CURREL] = 0;
01950     ACTIVE[.CHANNEL,SIZE] = .DESCRIPTOR[.CHANNEL,SIZE];
01960     ACTIVE[.CHANNEL,FILSTAT]<descno> = 0;
01970     ACTIVE[.CHANNEL,FILSTAT]<wrtbit> = 0;
01980     ACTIVE[.CHANNEL,FILSTAT]<curmod> = 0;
01990     ACTIVE[.CHANNEL,FILSTAT]<firstmod> = 0;
02000     ACTIVE[.CHANNEL,FILSTAT]<delbits> = 0;
02010     ACTIVE[.CHANNEL,BUFADDR] = .BUFPTR;
02020     ACTIVE[.CHANNEL,PARENT] = .DESCRIPTOR[.CHANNEL,CURDESC];
02030
02040 end;
02050
02060
02070
02080

```

02090
02100

000 31.0000

```

00010 MODULE PROTCHNCHECK(NOLIST)=
00020 begin
00030
00040 % this file contains a routine PROTCHK
00050 % to check for validity of access on a file %
00060
00070
00080 % author : AARTI KUMAR
00090 % date of creation: 15 JUNE, 1982 %
00100
00110
00120 MACRO
00130
00140 protection = 0,9$, !! file=protection specified,
00150 prog = 0,8$, !! programmer number of user's ppn
00160 prj = 0,8$, !! protection number of user's ppn
00170 prg = 0,3$, !! protection for all user's
00180 prj = 3,3$, !! protection for user's of same protect
00190 onr = 6,3$, !! protection for owner
00200 SYSPPN = #600401$, ! system ppn { #001, #001}
00210
00220
00230 GLOBAL ROUTINE PROTCHK(ppn,function,owner,lock) =
00240 begin
00250 Local VALUE,FLAG;
00260
00270 % this routine is called to do a protection check %
00280
00290
00300 if ( .ppn eq1 SYSPPN ) then RETURN 1 ;
00310 if .ppn<prj> eq1 .owner<prj> then RETURN 1 ;
00320 then if .ppn<prg> eq1 .owner<prg>
00330 then VALUE = .LOCK<onr>
00340 else VALUE = .LOCK<prj>
00350 else VALUE = .LOCK<prg>
00360 if .VALUE leq .FUNCTION then (RETURN 1 )
00370 else (RETURN 0);
00380
00390 end;
00400
00410 end ELUDOM
00420
00430

```

```

00010 % definitions of all variable declarations in disk manager %
00020
00030 % author : AARTI KUMAR ; 1 JULY ,1982 %
00040
00050
00060
00070
00080 external STATUS, LASTJOB;
00090 external REQUEST, BUFFER;
00100
00110 structure ARRAY[PTK, REC] = [num*ind*2] (.ARRAY + .PTR*ind*2 + .REC*2) < 0, 8*2 >;
00120
00130 map ARRAY REQUEST; ! REQUEST TABLE FOR DISK
00140
00150 map VECTOR BUFFER;
00160
00170 external STARTIO , AWAKE;
00180
00190 MACRO
00200
00210
00220
00230
00240
00250
00260
00270
00280
00290
00300
00310
00320
00330
00340
00350
00360
00370
00380
00390
00400
00410
00420
00430
00440
00450
00460
00470

```

IN == STATUS<0,4>\$,
OUT == STATUS<4,4>\$,
PENDING == STATUS<8,1>\$,
AWAKEN == STATUS<9,1>\$,
FLAG == STATUS<10,1>\$,
JOBNO == 0\$,
IOSTAT == 1\$,
BLKADD == 2\$,
LOCN == 3\$,
MAXM == 40\$,
DISKBLK == REQUEST[.OUT, BLOCK]\$,
LOCATION == REQUEST[.OUT, LOCN]\$,
operation == 0, 1\$,
extension == 14, 2\$,
ERRVAL == #077000\$,
CURBLK == 1\$,
CURSTAT == 0\$,
CURLOCN == 2\$,
WRDNO == 3\$;

```

00480 external DSKERR;
00490
00500 MACRO REMOVE(JOB, NUMB) = (NUMB + ERRVAL) ; RETURN )$;

```

```

00010  MODULE DIVIDE(MOLIST)=
00020  BEGIN
00030
00040  GLOBAL ROUTINE MODL(DIVIDEND,DIVISOR)=
00050  BEGIN
00060  WHILE .DIVIDEND GEO .DIVISOR DO
00070  BEGIN
00080  DIVIDEND = .DIVIDEND - .DIVISOR;
00090  END;
00100  DIVIDEND
00110  END;
00120
00130
00140  GLOBAL ROUTINE DIVL(DIVIDEND,DIVISOR) =
00150  BEGIN
00160  LOCAL QUOTIENT;
00170  QUOTIENT = 0;
00180  WHILE .DIVIDEND GEO .DIVISOR DO
00190  BEGIN
00200  DIVIDEND = .DIVIDEND - .DIVISOR; QUOTIENT = .QUOTIENT + 1;
00210  END;
00220  QUOTIENT
00230  END;
00240  END ELUDUM
00250

```

```
MODULE DISKMANAGER(MOLIST) =
begin
```

```
% DISK MANAGER FOR DISK I/O REQUESTS %
%
% AUTHOR : AARTI KUMAR
% DATE OF CREATION : 1 JULY, 1982 %
```

```
MACRO
```

```
IN = STATUS<0,4>$,
OUT = STATUS<4,4>$,
PENDING = STATUS<8,1>$,
AWAKEN = STATUS<9,1>$,
FLAG = STATUS<10,1>$,
JORNNO = 0$,
IOSTAT = 1$,
BLKADD = 2$,
LOCN = 3$,
latter = 15,1$,
MAXM = 40$,
DISKBLK = REQUEST[.OUT,BLKADD]$,
LOCATION = REQUEST[.OUT,LOCN]$,
ERRVAL = #077000$,
operation = 0,1$,
extension = 14,2$,
function = 0,3$,
CURBLK = 1$,
CURSTAT = 0$,
CURLOCN = 2$,
WRDNO = 3$;

! index to next free entry in table
! next request to be carried out
! {0/1 - interrupt pending no/yes}
! {0/1 - last job to be awakened yes/no}
! { descriptor read operation}
! user's job number field
! status word stored with each request
! block address for disk i/o
! memory location for descriptor in block
! position of requests possible
! number of requests possible
! points to block address in request
! points to memory location
! error value range starts
! {0/1-read/write range operation}
! { memory extension bits stored
! { operation specified by user}
! block address of current request
! status word of current request
! memory location for current request
! complement of number of words transferred
```

```
STRUCTURE TABLE[NO,FIELD] =
[NO*FIELD*BYTES](.TABLE + .NO * FIELD * BYTES + .FIELD * BYTES )<0,8*BYTES>;

word global TABLE REQUEST[40,4]; ! request table for all disk i/o
word global VECTOR BUFFER[128]; ! local buffer used for descriptor operations
external IOTABLE;
map VECTOR IOTABLE; ! stores current request
global DSKERR;
GLOBAL STATUS, ! status word for disk manager
LASTJOB; ! stores last job number
external INITIO,SETPRIOR,RESETPRIOR,BLOCK;
```

```
00010
00020
00030
00040
00050
00060
00070
00080
00090
00100
00110
00120
00130
00140
00150
00160
00170
00180
00190
00200
00210
00220
00230
00240
00250
00260
00270
00280
00290
00300
00310
00320
00330
00340
00350
00360
00370
00380
00390
00400
00410
00420
00430
00440
00450
00460
00470
00480
00490
00500
00510
00520
```



```

00530 MACRO OUTERR(NUMB)= (NUMB + ERKVAL) ; RETURN -1)$;
00540 ( USKERR =
00550
00560
00570
00580 GLOBAL ROUTINE STARTIO =
00590 begin
00600 local PTR ;
00610
00620
00630
00640
00650
00660
00670
00680
00690
00700
00710
00720
00730
00740
00750
00760
00770
00780
00790
00800
00810
00820
00830
00840
00850
00860
00870
00880
00890
00900
00910
00920
00930
00940
00950
00960
00970
00980
00990
01000
01010
01020
01030
01040

% this routine is called to initiate the operation specified
% by the user %

case .REQUEST[.OUT,IOSTAT]<function> of
set
!0 - READ BLOCK
(AWAKEN = 1 ;
IOTABLE[CURSTAT]<operation> = 0 ;
IOTABLE[CURSTAT]<extension> = .REQUEST[.OUT,IOSTAT]<extension> ;
IOTABLE[CURBLK] = .DISKBLK ;
IOTABLE[CURLOCN] = .LOCATION ;
IOTABLE[WRDNO] = -128 ;
INITIO() ) ;

!1 - WRITE BLOCK
(AWAKEN = 1 ;
IOTABLE[CURSTAT]<operation> = 1 ;
IOTABLE[CURSTAT]<extension> = .REQUEST[.OUT,IOSTAT]<extension> ;
IOTABLE[CURBLK] = .DISKBLK ;
IOTABLE[CURLOCN] = .LOCATION ;
IOTABLE[WRDNO] = -128 ;
INITIO() ) ;

!2- READ DESCRIPTOR
(AWAKEN = 0 ; REQUEST[.OUT,IOSTAT] = 4 ;
IOTABLE[CURSTAT]<operation> = 0 ;
IOTABLE[CURBLK] = .DISKBLK ;
IOTABLE[CURLOCN] = .BUFFER ;
IOTABLE[WRDNO] = -128 ;
OUT = .OUT - 1 ;
INITIO() ) ;

!3 - WRITE DESCRIPTOR
(AWAKEN = 0 ; REQUEST[.OUT,IOSTAT] = 5 ;
IOTABLE[CURSTAT]<operation> = 0 ;
IOTABLE[CURBLK] = .DISKBLK ;
IOTABLE[CURLOCN] = .BUFFER ;
IOTABLE[WRDNO] = -128 ;

```

```

01570 RESEIPRIOR();
01580 end;
01590
01600
01610 GLOBAL ROUTINE INTSTAT=
01620 begin
01630
01640
01650
01660
01670
01680
01690
01700
01710
01720
01730
01740
01750
01760
01770
01780
01790
01800
01810
01820
01830
01840
01850
01860
01870

    % initializes the disk manager %

    Incr I from 0 to 39 do
    begin
        Incr J from 0 to 3 do
        begin
            REQUEST[I..J] = 0;
        end;
    end;

    Incr I from 0 to 127 do ( BUFFER[I] = 0 ) ;
    STATUS = 0; LASTJOB = 0;
    DSKERR = 0;
    Incr I from 0 to 3 do ( IOTABLE[I] = 0 );

end;

end ELUDOM

```

```

00010 MODULE DSKDRIVER(NOLIST)=
00020 begin
00030
00040
00050
00060
00070
00080
00090
00100
00110
00120
00130
00140
00150
00160
00170
00180
00190
00200
00210
00220
00230
00240
00250
00260
00270
00280
00290
00300
00310
00320
00330
00340
00350
00360
00370
00380
00390
00400
00410
00420
00430
00440
00450
00460
00470
00480
00490
00500
00510
00520

% DISK DRIVER %
%
% AUTHOR : AARTI KUMAR %
% DATE OF CREATION : 2 JULY, 1982 %

MACRO

    retries = 0,4$,
    num = 40$,
    ind = 4$,
    errbit = 15,1$,
    SECSIZE = 128$,

    ! number of retries stored
    ! maximum number of requests
    ! no of fields in each request
    ! error condition indicator size
    ! 2's complement of sector size

% command bits in the command register %
    seekbit = 0,1$,
    serbit = 1,1$,
    wrtbit = 3,1$,
    readbit = 2,1$,
    enabl = 6,1$,
    headadv = 7,1$,
    memextn = 8,2$,
    unit = 10,1$,
    port = 11,2$,
    varb = 13,1$,
    mask = 14,1$,

    ! {0/1 = seek command off/on}
    ! search command specified
    ! write command specified
    ! read command specified
    ! interrupt enabled
    ! head advance command
    ! memory extension specified
    ! unit of disk drive
    ! port-{01}
    ! {no of words variable}
    ! interrupt mask bit

% field specifications in disk block %
    trk = 0,8$,
    sectr = 0,5$,
    surfc = 5,4$,

    ! track field in drive register
    ! sector field in drive register
    ! surface field in drive register

% bits in control and status register %
    unsafe = 8,1$,
    done = 14,1$,
    conready = 7,1$,
    crccerr = 2,1$,
    busy = 0,1$,
    nonmem = 12,1$,

    ! drive unsafe
    ! operation done
    ! controller ready
    ! crcc error in write
    ! controller busy
    ! error indicates wrong memory

% bits in attention register %
    port1 = 3,1$,
    port2 = 2,1$,

    ! port 1 specified
    ! port 2 specified

```



```

01050 TEMP = 0;
01060 TEMP<unit> = 0;
01070 TEMP<port> = 1;
01080 TEMP<enabl> = 1;
01090 TEMP<sekbitt> = 1;
01100 TEMP<serbitt> = 1;
01110 TEMP<wrtbitt> = 1;
01120
01130 ends;
01140
01150 global TEMP,ERROR;
01160 word global VECTOR IOTABLE[4];
01170
01180 require DISK1.B11;
01190
01200 external DIVL,MODL;
01210
01220 GLOBAL ROUTINE INITIU=
01230 begin
01240
01250 % this routine initiates a disk read/write
01260 % by entering in all controller registers appropriate values &
01270
01280 TEMP=0; ERROR<errbitt>=0;
01290 until .CNTRLSTAT<conready> eql 1 do ;
01300   TRKREG<trk> = IOTABLE[CURBLK]<cylinder>;
01310   BLKREG<sectr> = MODL( IOTABLE[CURBLK]<filblk> , 10) + 1;
01320   BLKREG<surfc> = DIVL( IOTABLE[CURBLK]<filblk> , 10 );
01330   WRDCNT = IOTABLE[WRDNO];
01340   ADDRNG = IOTABLE[CURLOCN];
01350   case set
01360   of
01370   !0- read
01380   ( READCOMMAND;
01390     TEMP<memextn> = IOTABLE[CURSTAT]<extension> ;
01400     if IOTABLE[WRDNO] lss SECsize then TEMP<varb> = 1;
01410     COMREG = .TEMP );
01420   !1-write
01430   ( WRTCOMMAND;
01440     TEMP<memextn> = IOTABLE[CURSTAT]<extension> ;
01450     if IOTABLE[WRDNO] lss SECsize then TEMP<varb> = 1;
01460     COMREG = .TEMP );
01470   tes;
01480
01490 end;
01500
01510
01520
01530
01540
01550

```



```

02090
02100
02110
02120
02130
02140
02150
02160
02170
02180
02190
02200
02210
02220
02230
02240
02250
02260
02270
02280
02290
02300
02310
02320
02330
02340
02350
02360
02370
02380
02390
02400
02410
02420
02430
02440
02450
02460

begin
% interrupt servicing for non data commands %

if .ATINREG<port2> then
begin
COMREG = SENSEF;
if not .DRVSTAT<ready> then
begin
COMREG = RESTORE; ERROR<retries> + 1;
ERROR<retries> = .ERROR<retries> eq 5 then
if .ERROR<retries> eq 1; ERROR<RETRIES> = 0 ) else INITIO();
( ERROR<errorbit> = 0;
end
end;

end;

GLOBAL ROUTINE INITDISK=
begin
% initializes disk controller %

DTVT = DATATRANSFER;
(DTVT + 2) = #340;
NDTVT = NONDATA;
(NDTVT + 2) = #340;
DO ( COMREG = #104000) UNTIL .DRVSTAT<ONLINE> EOL 1;

end;

end ELUDOM

```

```

00010 MODULE FILLBUFFERS( NOLIST) =
00020 begin
00030 % LINE PRINTER BUFFER MANAGER %
00040
00050 AUTHOR : AARTI KUMAR
00060 % DATE OF CREATION : JULY 4, 1982 %
00070
00080
00090
00100
00110
00120 MACRO
00130 EOLN = #12$,
00140 RINGSIZE = #15$,
00150 TYPE: = #0$,
00160 LINK = #1$,
00170 EM = #0$,
00180 IN = #1$,
00190 OUT = #2$,
00200 waiton LPTSTAT<10,1>$,
00210 pending LPTSTAT<11,1>$,
00220 intron LPTSTAT<12,1>$,
00230 BLANK = #40$,
00240 MSG1 LPTSTAT<0,5>$,
00250 CEM LPTSTAT<5,5>$,
00260
00270
00280
00290
00300 global LPTSTAT, ! current status of all buffers and driver
00310 INPTR, ! pointer to next character in current buffer
00320 BANRPTR, ! ptr to buffer containing banner
00330
00340 structure KING(no,cnt) = [no*cnt*bytes](<RING + no*cnt*bytes + cnt*bytes>,<0,8*bytes>;
00350 structure LNK(RNGH,K) = [h*k*bytes](<LNKRNG + h*k*bytes + k*bytes>,<0,8*bytes>;
00360
00370 byte global RING LPTBUF[15,134]; ! ring of one line LPTBUFS
00380 byte global LNK(RNG HSI[15,2]); ! current status of each buffer
00390 byte global VECTOR FI[2]; ! ptr to first of each type
00400 byte global VECTOR CUR[4]; ! ptr to current of each type
00410
00420 external FILERUF; ! file buffer for currently opened file
00430 map VECTOR FILERUF;
00440
00450 external IDENT; ! stores ascii form of banner
00460 map VECTOR IDENT;
00470
00480 external CURPTR; ! ptr to next character in file buffer
00490 external PUTLINE;
00500 external PSEM,VSEM; ! counting semaphores
00510 external SETPPRIOR,RESETPPRIOR;

```

! no of buffers available
 ! type of buffer field
 ! ptr to next of similar type
 ! empty {buffer status}
 ! INput {buffer status}
 ! output {buffer status}
 ! bit indicates file block wait
 ! bit indicates interrupt pending
 ! bit indicates interrupt servicing is on
 ! blank character
 ! wait for next file block
 ! count of no of empty buffers
 ! count of no of INout full buffers


```

00530 GLOBAL routine BUNIT=
00540 begin
00550   ! this is an initializing procedure for the driver %
00560   Local I;
00570
00580   LPTSTAT = 0;
00590   BANRPTR = -1;
00600   INPTR = 0;
00610   CEM = RNGSIZE;
00620   CIN = 0;
00630   F[EM] = RNGSIZE -1;
00640   F[IN] = -1;
00650   CUR[EM] = 0;
00660   CUR[IN] = -1;
00670   CUR[OUT] = -1;
00680   I = RNGSIZE -1;
00690   do begin
00700     BSI, I, LINKJ = .I-1;
00710     I = I -1;
00720     until (.I eq 0);
00730     INCR J from 0 to (RNGSIZE-1) do
00740       (BSI, J, TYPEJ = EM);
00750       pending = 0;
00760       waiton = 0; intron = 0;
00770       INCR J from 0 to 14 do
00780         begin
00790           Incr I from 0 to 133 do
00800             begin
00810               LPTBUF[.J , .LJ] = 0 ;
00820             end;
00830           end;
00840         end;
00850       end;
00860
00870   end;
00880
00890 end;
00900
00910 global routine TAKEBUF(KIND)=
00920 begin
00930   Local TEMP, VALUE;
00940   VALUE = .FI.KINDJ;
00950   TEMP = .BSI.FI.KINDJ , LINKJ;
00960   FI.KINDJ = .TEMP;
00970   .VALUE
00980
00990   ! get next buffer of type "KIND"
01000

```

```

01050 Local TEMP;
01060 TEMP = .CURF.KIND1;
01070 BSL.TEMP,LINE1 = .NUM;
01080 BSL.NUM,TYPE1 = .KIND;
01090 BSL.NUM,TYPE1 = .KIND;
01100 BSL.NUM,TYPE1 = .KIND;
01110 CURF.KIND1 = .NUM;
01120
01130 end;
01140
01150 GLOBAL routine SENDLINE=
01160 begin
01170 % this is a procedure for sending a line at a time
01180 % to one of a ring of buffers %
01190
01200 SETPRIOR();
01210 If not .waiton then
01220 begin
01230 INPTR = 0;
01240 PSEM(EM,10);
01250 CURF1 = TAKEBUF(EM);
01260
01270 end;
01280 while (.FILEBUF1.CURPTR) neq EOLN)
01290 do begin
01300 LPTBUF1.CURF1, INPTR = .FILEBUF1.CURPTR;
01310 INPTR = .INPTR + 1;
01320 CURPTR = .CURPTR + 1;
01330 If ( .CURPTR eq 256 ) then ( waiton = 1; RETURN MSG1);
01340 ! block { wait for disk I/O }
01350 end;
01360 LPTBUF1.CURF1, .INPTR = EOLN;
01370 waiton = 0;
01380 ADDBUF( .CURF1, IN);
01390 VSEM( IN,11);
01400 If not .pending then ( intron = 0; PUTLINE() );
01410 RESETPRIOR();
01420
01430 i
01440 end;
01450
01460 GLOBAL routine BANNER=
01470 begin
01480 Local PTR;
01490
01500 % printing of banners %
01510
01520

```

```

01570 BANRPTR = .CURLIN; do (LPTBUF(.CURLIN),.I) = BLANK );
01580 Incr I from 0 to 5; PTR = 0;
01590 While ( PTR = 6; PTR = 7 ) do
01600 ( LPTBUF(.CURLIN),.I) = IDENT(.PTR);
01610 LPTBUF(.CURLIN),.I) = .INPTR + 1; PTR = .PTR + 1;
01620 ADDBUF(.CURLIN),.I);
01630 VSEM( IN,11);
01640 If not pending then (intron = 0 ; PUTLINE() );
01650 RESETPRTOR();
01660
01670
01680
01690
01700
01710
01720
01730
01740
01750

```

end;

end ELUDOM

```

00010 MODULE LPTSPooler(NOLIST)=
00020 begin
00030
00040 % LINE PRINTER SPOOLER PROGRAM %
00050
00060 % AUTHOR : AARTI KUMAR
00070 % DATE OF CREATION : JULY 2, 1982 %
00080
00090
00100 % THIS FILE CONTAINS TWO MAIN ROUTINES
00110 (1) ENTERSPOOL: this routine enters print requests
00120 in the queue maintained by the spooler
00130 (2) LPTSPL: this routine runs as a user process
00140 which accesses the next request in the queue
00150 (for printing) and opens and transfers
00160 the file to the printer %
00170
00180
00190
00200 MACRO
00210 MAXSEQ = 1000$, ! MAX-limit of sequence no
00220 INITSEQ = 100$, ! min-limit of sequence no
00230 SEQNO = 0$, ! field specification for seq-no's in spool table
00240 JOBNO = 1$, ! field specification for job-no's in spool table
00250 FNAME = 2$, ! pointer to filename
00260 BNAME = 6$, ! field specification for user's pbn
00270 PAGE = 7$, ! field specification for page-limit
00280 SIN = 0$, ! pointer to next free entry in table
00290 PRNTON = 6>$, ! pointer to entry for next print request
00300 EOLIST = 15>$, ! {0/1- FILE PRINTING OFF/ON }
00310 MSG1 = 1$, ! end of list marker
00320 MAXM = 50$, ! size of spooler table
00330 SPLJOB = 50$, ! job number of spooler
00340 EOF = 1$, ! end of file reached
00350
00360
00370
00380 % format of file name : [direc-addr][filename][filename][eolist][4 words] %
00390 % mode of access of requests : [index, <field-specification>] %
00400
00410 structure TABLE[NO, FIELD] = [NO*FIELD*BYTES](.TABLE + .NO*FIELD*BYTES + .FIELD*BYTES)<0, 8*BY
00420
00430 word global TABLE SPOOL[50, 8]; ! table to maintain a queue of all print requests
00440 byte global VECTOR FILEBUF[256]; ! buffer to transfer blocks of open file
00450
00460 global CURPTR, ! pointer to next character in buffer
00470 POINTER, ! status word to store IN and OUT pointers of table
00480
00490

```

```

00530 external SENDLINE;      ! routine to send a line to a ring of one line buffers
00540 external RANNER;      ! routine to print banners for a file
00550 external ABORT;      ! routine to abort jobs
00560 external OPEN, GET, CLOSE;      ! routines to open and read file blocks
00570 external BLOCK, AWAKE, SETPRIOR, RESETPRIOR;
00580
00590 routine SEQ=
00600 begin
00610   LASTSEQ = LASTSEQ + 1;
00620   if ( .LASTSEQ eq1 MAXSEQ ) then (LASTSEQ = INITSEQ);
00630   .LASTSEQ
00640 end;
00650
00660 byte global VECTOR IDENT[8];
00670
00680 routine CNVRTOCT=
00690 begin
00700   IDENT[0] = .SPOOL[.SOUT,BANR]<14,2> + #60 ;
00710   IDENT[1] = .SPOOL[.SOUT,BANR]<11,3> + #60 ;
00720   IDENT[2] = .SPOOL[.SOUT,BANR]<8,3> + #60 ;
00730   IDENT[3] = #54;
00740   IDENT[4] = .SPOOL[.SOUT,BANR]<6,2> + #60 ;
00750   IDENT[5] = .SPOOL[.SOUT,BANR]<3,3> + #60 ;
00760   IDENT[6] = .SPOOL[.SOUT,BANR]<0,3> + #60 ;
00770
00780 end;
00790
00800 global routine INITSPOOL= ! this returns the next sequence number available
00810 begin
00820   LASTSEQ = 99;
00830   SIN = 0; SOUT = 0;
00840   POINTER = 0;
00850   PRNTDN = 0;
00860   Incr I from 0 to 49 do
00870     begin
00880       Incr J from 0 to 7 do
00890         begin
00900           SPOOL[.I , .J] = 0 ;
00910         end;
00920       end;
00930     end;
00940   end;
00950
00960
00970
00980
00990
01000

```

```

01050 % this routine enters requests in the spooler table %
01060
01070
01080
01090
01100
01110
01120
01130
01140
01150
01160
01170
01180
01190
01200
01210
01220
01230
01240
01250
01260
01270
01280
01290
01300
01310
01320
01330
01340
01350
01360
01370
01380
01390
01400
01410
01420
01430
01440
01450
01460
01470
01480
01490
01500
01510
01520

SETPRIOR();
If ( (.SIN eql .SOUT ) and .PRNTON ) then ABORT(.JOB,40); ! table full
  SPOOL(.SIN,SEOND ) = SEQ();
  SPOOL(.SIN,JOURNO ) = .JOB;TH);
  SPOOL(.SIN,FNAME ) = (.PATH + 2);
  SPOOL(.SIN,FNAME+1) = (.PATH + 4);
  SPOOL(.SIN,FNAME+2) = .COLIST;
  SPOOL(.SIN,FNAME+3) = .PPN;
  SPOOL(.SIN,PAGE ) = .LIM;
  SIN = SIN + 1;
  if ( .SIN eql MAXM ) then (SIN = 0);
  if not .PRNTON then AWAKE();
  RESETPRIOR();
end;

routine TRANSFER=
begin
% this routine transfers a single file block to a ring
% of buffers maintained by the spooler for printing lines %
Local VALUE;
CURPTR = 0;
While ( .CURPTR lss 256 ) do
  begin
    VALUE = SENDLINE();
    if ( .value eql MSG1 ) then RETURN ;
    CURPTR = .CURPTR + 2;
    if ( .FILEBUF.CURPTR eql #14 ) then RETURN ;
  end;
end;

OWN VAL2;
WORD OWN VECTOR VAL[4] ;

ROUTINE NEXTOUT=
begin
% this routine takes the next request for printing
% from the queue %

```

```

01570 INCR I FROM 0 TO 3 DO ( VAL[.I] = .SPOOL[.SOUT,FNAME + .I] ) ;
01580 VAL2 = .SPOOL[.SOUT,BANNER] ;
01590 CNVRTOCT();
01600 RESETPRIOR();
01610
01620 end;
01630
01640 ROUTINE INCROUT=
01650 begin
01660 % this increments the out pointer %
01670
01680 SETPRIOR();
01690 PRNTOUT = 0;
01700 SOUT = .SOUT + 1;
01710 if ( .SOUT eq1 MAXM ) then ( SOUT = 0 ) ;
01720 RESETPRIOR();
01730
01740 end;
01750
01760 GLOBAL ROUTINE LPTSPL=
01770 begin
01780 % this routines runs as a user process and repeatedly opens
01790 % and transfers the next file in the table for printing %
01800
01810 while 1 do
01820 begin
01830 NEXTOUT();
01840 CHANL = OPEN(SPLJOB, .VAL2, VAL, 0, FILEBUF);
01850 BANNER();
01860 do begin
01870 FILBLK = GET(SPIJOB, .CHANL);
01880 TRANSFER();
01890
01900 end
01910 until .FILBLK eq1 EOF;
01920 CLOSE(SPLJOB, .CHANL);
01930 INCROUT();
01940
01950 end;
01960
01970 end;
01980
01990
02000
02010
02020
02030
02040
02050 end ELUDOM

```



```

00530 GLOBAL CURCOND,LCOMND ;
00540
00550 external TAKEBUF, ADDRBUF ;
00560 external LPTSTAT,BANRPTR;
00570 external LPTBUF,CUR;
00580 external PSEM,VSEM;
00590
00600 structure ARY[no,ind] = [15*134*BYTES](.ARY + .no*134*BYTES + .ind*BYTES)<0,8*BYTES>;
00610
00620 map ARY LPTBUF;
00630 map VECTOR CUR;
00640
00650
00660 GLOBAL ROUTINE PRNTLINE=
00670 begin
00680
00690 % this routine initiates the printing of a line %
00700
00710 Local TMP1,TMP2;
00720
00730 TMP1 = 0; TMP2 = .LPTRUF[.CUR[OUT] , .TMP1];
00740 while ( .TMP2 neq EOLN ) do
00750 begin
00760 until .LCSR<done> do ;
00770 if .TMP2<vfulline> then ( TMP2<imdt> = 0 );
00780 LPBR<data> = .TMP2<data>;
00790 TMP1 = .TMP1 + 1;
00800 TMP2 = .LPTRUF[.CUR[OUT] , .TMP1] ;
00810 end;
00820
00830 end;
00840
00850 GLOBAL ROUTINE PRNTBANR=
00860 begin
00870 Local TMP1 , TMP2;
00880
00890 % Initiates printing of banner %
00900
00910 TMP1 = 0; TMP2 = .LPTRUF[.CUR[OUT] ,.TMP1];
00920 incr I from 0 to 9 do
00930 begin
00940 until .LCSR<done> do;
00950 LPBR<data> = BLANK;
00960 end;
00970 while ( .TMP2 neq EOLN ) do
00980 begin
00990 until .LCSR<done> do ;
01000 if .TMP2<vfulline> then ( TMP2<imdt> = 0 );
01010 LPBR<data> = .TMP2<data>;
01020 TMP1 = .TMP1 + 1;
01030 TMP2 = .LPTRUF[.CUR[OUT] , .TMP1] ;
01040 end;

```

```

01050
01060
01070
01080
01090
01100
01110
01120
01130
01140
01150
01160
01170
01180
01190
01200
01210
01220
01230
01240
01250
01260
01270
01280
01290
01300
01310
01320
01330
01340
01350
01360
01370
01380
01390
01400
01410
01420
01430
01440
01450
01460
01470
01480
01490
01500
01510
01520
01530
01540
01550
01560

end;

GLOBAL ROUTINE POSTBANR=
begin
% prints 10 blank lines after banner %
until .LCSR<ready> do;
until .LCSR<done> do;
BANRPTR = -1;
LVFR<vfulline> = 1;
LVFR<imdt> = 1;
LVFR<vfucm> = 10;
CURCOND = dummy;
LCOMND = 0;
LCOMND<prntcom> = 1;
LCOMND<enabl> = 1;
LCSR = .LCOMND;

end;

GLOBAL ROUTINE PRNTDUMMY=
begin
% prints a dummy line to shift control to next line %
until .LCSR<ready> do;
until .LCSR<done> do;
LCOMND = 0;
CURCOND = line;
LCOMND<prntcom> = 1;
LCOMND<enabl> = 1;
LCSR = .LCOMND;

end;

GLOBAL ROUTINE POSTEJECT=
begin
% leaves a margin of 5 lines after paper eject %
until .LCSR<ready> do;
until .LCSR<done> do;
LVFR<vfulline> = 1;
LVFR<imdt> = 1;
LVFR<vfucm> = 5;
if ( .CUR[OUT] eq1 .BANRPTR ) then
( PRNTBANR();
CURCOND = banron )
else ( PRNTLINE(); CURCOND = dummy );
LCOMND = 0;

```

```

01570 LCOMMD<enabl> = 1;
01580 LCOMMD<prntcom> = 1;
01590 LCSR = .LCOMMD;
01600
01610 end;
01620
01630 GLOBAL ROUTINE PAGEJECT=
01640 begin
01650   until .LCSR<ready> do;
01660     until .LCSR<done> do;
01670       LCOMMD = 0;
01680       LCOMMD<enabl> = 1;
01690       LCOMMD<eject> = 1;
01700       CURCOND = p1eject;
01710       LCSR = .LCOMMD;
01720
01730
01740
01750
01760
01770
01780 GLOBAL ROUTINE PUTLINE=
01790 begin
01800
01810   % initiates printing %
01820
01830   if not PSEM(IN,11) then RETURN ;
01840   CUR[OUT] = TAKEBUF(IN);
01850   if ( (.CUR[OUT] eq1 .BANKPTR ) or
01860     ( .CCSR<linecnt> eq1 31 ) ) then
01870     begin PAGEJECT();
01880     pending = 1;
01890   end
01900
01910   else begin
01920     until .LCSR<ready> do ;
01930     until .LCSR<done> do;
01940       PRNTLINE();
01950       LCOMMD = 0;
01960       LCOMMD<prntcom> = 1;
01970       LCOMMD<enabl> = 1;
01980       pending = 1;
01990       CURCOND = line;
02000       LCSR = .LCOMMD;
02010
02020
02030
02040
02050 end;
02060
02070 GLOBAL ROUTINE INTERRUPT PRINTER=
02080 begin

```

```

02090 case .CURCOND of
02100 set
02110 !0 - pject
02120 { POSTJECT() };
02130 !1 - banron
02140 { POSTBANP() };
02150 !2 - dummy
02160 { PRNTDUMMY() };
02170
02180 !3 - line
02190 { pending = 0;
02200   ADDBUF( .CUR[OUT] , EM);
02210   VSEM(EM ,10);
02220   intron = 1 ; PULLINE() };
02230
02240
02250 tes;
02260
02270 end;
02280
02290 ROUTINE INTERRUPT ERRPRNT=
02300 begin
02310
02320 end;
02330
02340
02350 GLOBAL ROUTINE INITPRNT=
02360 begin
02370
02380 % initialize the printer %
02390
02400 REDYINTR = PRINTER;
02410 ( REDYINTR + 2 ) = #340;
02420 ERRINTR = ERRPRNT;
02430 ( ERRINTR + 2 ) = #340;
02440
02450 end;
02460
02470
02480
02490 end ELUDOM
02500

```

```

00010 MODULE COUNTINGSEMAPHORES(NOLIST)=
00020 begin
00030
00040 % COUNTING SEMAPHORE IMPLEMENTATION %
00050
00060 % AUTHOR : AARTI KUMAR
00070 % DATE OF CREATION : 19 JULY,82 %
00080
00090
00100
00110
00120 MACRO CEM = LPTSTAT<0,5>$, ! count of empty buffers
00130 CIN = LPTSTAT<5,5>$, ! count of full buffers
00140 Intron = LPTSTAT<12,1>$, ! interrupt servicing is on/off
00150
00160 external LPTSTAT;
00170 external AWAIT,SIGNAL; ! synchronizing routines
00180
00190
00200
00210 GLOBAL routine PSEM(KIND,EVENT)=
00220 begin
00230 Local MUTEX;
00240
00250 % this routine decrements the semapSEMore
00260 and does a wait if value is less then 0 %
00270
00280
00290 case ,KIND of
00300 set
00310 !0= empty
00320 { MUTEX = .CEM };
00330 !1= full
00340 { MUTEX = .CIN };
00350 tes;
00360 If ( .MUTEX eq 0 ) then
00370 ( ( if not .intron then ( AWAIT(.EVENT) )
00380 else ( RETURN 0 ) )
00390 else ( MUTEX = .MUTEX - 1 );
00400 case ,KIND of
00410 set
00420 !0= empty
00430 { CEM = .MUTEX };
00440 !1=full
00450 { CIN = .MUTEX };
00460 tes;
00470
00480
00490
00500
00510
00520
00530
00540
00550
00560
00570
00580
00590
00600
00610
00620
00630
00640
00650
00660
00670
00680
00690
00700
00710
00720
00730
00740
00750
00760
00770
00780
00790
00800
00810
00820
00830
00840
00850
00860
00870
00880
00890
00900
00910
00920
00930
00940
00950
00960
00970
00980
00990
01000
end;

```

```

00530 begin
00540
00550 % this routine increments the semapsemore
00560 % and signals the job waiting on this event %
00570
00580 case ,KIND of
00590 set
00600 !0= empty
00610 ( CEM = .CEM + 1 );
00620 !1= full
00630 ( CIN = .CIN + 1 ) ;
00640 tes;
00650 SIGNAL(.EVENT);
00660
00670 end;
00680
00690
00700 end ELUDOM
00710

```